NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ADAPTATION OF A FAULT–TOLERANT FPGA–BASED LAUNCH SEQUENCER AS A CUBESAT PAYLOAD PROCESSOR

by

Jordan K. Goff

June 2014

Thesis Co-Advisors:	Herschel H. Loomis, Jr. James H. Newman

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE					*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.					
1. AGENCY USE ONLY *(Leave blank)*		**2. REPORT DATE** June 2014	**3. REPORT TYPE AND DATES COVERED** Master's Thesis		
4. TITLE AND SUBTITLE ADAPTATION OF A FAULT–TOLERANT FPGA–BASED LAUNCH SEQUENCER AS A CUBESAT PAYLOAD PROCESSOR					**5. FUNDING NUMBERS**
6. AUTHOR(S) Jordan K. Goff					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000					**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A					**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.					
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited					**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

The purpose of this thesis is to design and test a fault–tolerant reduced instruction set computer processor running a subset of the multiprocessor without interlocked pipelined stages instruction set. This processor is implemented on a field programmable gate array (FPGA) and will be used as the foundation for a payload processor on a cube satellite developed at the Naval Postgraduate School.

This thesis begins by considering the radiation effects present in the space environment and the various fault–tolerant designs used to guard against specific types of particle events. The internal triple modular redundancy method is selected and implemented at each pipeline stage of the processor. Next, a target FPGA is selected based on the performance requirements of the processor. The Virtex–5 (registered trademark of Xilinx, Inc.) is selected over the ProASIC3 (registered trademark of Microsemi, Inc.) due to its enhanced capabilities and potential to support expansion for future applications.

The hardware design is presented as a hybrid Verilog and schematic based design. The system consists of the processor and a universal asynchronous receiver/transmitter that reads and writes data received from a generic serial interface. The device is simulated to ensure proper logic functionality. Conclusions and future work are discussed.

14. SUBJECT TERMS Fault-tolerant, Payload Processor, Cube Satellite, Internal Triple Modular Redundancy, Field Programmable Gate Array, Universal Asynchronous Receiver Transmitter			**15. NUMBER OF PAGES** 289
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

ADAPTATION OF A FAULT–TOLERANT FPGA–BASED LAUNCH SEQUENCER AS A CUBESAT PAYLOAD PROCESSOR

Jordan K. Goff
Lieutenant, United States Navy
B.S., Rensselaer Polytechnic Institute, 2007

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2014

Author: Jordan K. Goff

Approved by: Herschel H. Loomis, Jr.
Thesis Co-Advisor

James H. Newman
Thesis Co-Advisor

Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The purpose of this thesis is to design and test a fault–tolerant reduced instruction set computer processor running a subset of the multiprocessor without interlocked pipelined stages instruction set. This processor is implemented on a field programmable gate array (FPGA) and will be used as the foundation for a payload processor on a cube satellite developed at the Naval Postgraduate School.

This thesis begins by considering the radiation effects present in the space environment and the various fault–tolerant designs used to guard against specific types of particle events. The internal triple modular redundancy method is selected and implemented at each pipeline stage of the processor. Next, a target FPGA is selected based on the performance requirements of the processor. The Virtex–5 (registered trademark of Xilinx, Inc.) is selected over the ProASIC3 (registered trademark of Microsemi, Inc.) due to its enhanced capabilities and potential to support expansion for future applications.

The hardware design is presented as a hybrid Verilog and schematic based design. The system consists of the processor and a universal asynchronous receiver/transmitter that reads and writes data received from a generic serial interface. The device is simulated to ensure proper logic functionality. Conclusions and future work are discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION AND OBJECTIVES 1
- **A. PURPOSE 1**
- **B. PREVIOUS WORK 3**
  - **1. NPSCuL Development 3**
  - **2. Launch Sequencer 5**
  - **3. Fault–tolerant, Pipelined Processor 6**
- **C. DESIGN METHODOLOGY AND THESIS ORGANIZATION 6**

II. RADIATION HARDENING AND FAULT–TOLERANCE 9
- **A. RADIATION HARDENING 9**
  - **1. Single–Event Effect 9**
  - **2. Single–Event Upset 9**
  - **3. Single–Event Transient 10**
  - **4. Single–Event Latch–up 10**
  - **5. Single–Event Burnout 10**
- **B. FAULT–TOLERANCE METHODS 11**
  - **1. Triple Modular Redundancy 11**
    - ***a. Block TMR 11***
    - ***b. Local TMR 12***
    - ***c. Global TMR 13***
    - ***d. Internal TMR 14***
  - **2. Other Fault–tolerance Approaches 15**
  - **3. Selection of a Fault–Tolerant Scheme for the Payload Processor 16**
- **C. CHAPTER SUMMARY 16**

III. DESIGN REQUIREMENTS AND SELECTION OF AN FPGA DEVICE 19
- **A. INTRODUCTION 19**
- **B. PERFORMANCE REQUIREMENTS 20**
  - **1. CLB Capability 20**
  - **2. RADHARD Requirements 21**
  - **3. Memory Requirements 22**
  - **4. Clocking Requirements 23**
  - **5. I/O Requirements 24**
  - **6. SWAP Requirements 24**
- **C. CANDIDATE FPGA DEVICES 25**
  - **1. Actel ProASIC3 ..... 27**
    - ***a. Architecture 28***
    - ***b. Radiation Tolerance 29***
    - ***c. Memory Capabilities 29***
    - ***d. I/O Capabilities 30***
    - ***e. Clock Management 31***
    - ***f. Power Consumption and Distribution 31***

2. **Xilinx Virtex–5** **32**
    a. ***Architecture*** ***32***
    b. ***Radiation Tolerance*** ***34***
    c. ***Memory Capabilities*** ***35***
    d. ***I/O Capabilities*** ***35***
    e. ***Clock Management*** ***37***
    f. ***Power Consumption and Distribution*** ***37***
D. **FPGA SELECTION** **37**
E. **CHAPTER SUMMARY** **38**

IV. DESIGN OF THE PAYLOAD PROCESSOR **39**
A. **PIPELINED PROCESSOR COMPONENTS AND ORGANIZATION** **39**
B. **PAYLOAD PROCESSOR DESIGN** **40**
1. **IF Stage** **41**
2. **ID Stage** **42**
3. **EX Stage** **46**
4. **MEM Stage** **48**
5. **WB Stage** **52**
C. **UART DESIGN** **53**
1. **Concept** **53**
2. **Receive Logic** **54**
3. **Transmit Logic** **55**
4. **Interface with the Payload Processor** **55**
D. **CHAPTER SUMMARY** **57**

V. TESTING AND ANALYSIS OF THE PAYLOAD PROCESSOR AND UART **59**
A. **PIPELINE REGISTERS** **59**
B. **TEST PROGRAM EXECUTION** **60**
1. **I–Format Instructions** **62**
2. **R–Format Instructions** **65**
3. **Load Word and Store Word Instructions** **68**
4. **Branch Instructions** **72**
5. **Jump Instructions** **77**
C. **UART TESTING** **79**
1. **UART Receive Testing** **79**
2. **UART Transmit Testing** **82**
D. **TROUBLESHOOTING** **85**
E. **CHAPTER SUMMARY** **87**

VI. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK **89**
A. **CONCLUSIONS** **89**
B. **FUTURE WORK** **90**
1. **Hazard Detection and Exception Handling** **90**
2. **Assembler, Compiler, and ISA Expansion** **91**
3. **Memory Management** **91**
4. **Implementation and Testing** **91**

5. Hardware Production 92
6. Radiation Testing 92
C. CLOSING REMARKS 93

APPENDIX A. VERILOG CODE AND SCHEMATICS 95
A. PAYLOAD PROCESSOR SYSTEM, PIPELINE, AND UART 96
B. PC REGISTER 98
C. IF STAGE 99
1. New PC Adder Module 99
2. Instruction Memory 100
3. Jump Address Calculator 102
D. IF/ID REGISTER 103
E. ID STAGE 104
1. Control Module 107
2. Register File 116
3. Sign Extension Module 119
F. ID/EX REGISTER 119
G. EX STAGE 123
1. ALU (ALU_bhv) 128
2. Address Adder (addr_adder) 130
3. Second ALU Operand Multiplexer (multiplex_2to1_nbit) 130
4. Immediate Offset Shifting Module (shift_left2) 130
5. Register Address Multiplexer (multiplex_2to1_reg_addr) 131
H. EX/MEM REGISTER 131
I. MEM STAGE 135
1. Branch Multiplexer (branch_multiplexer) 139
2. Address Selector Module (addr_sel_module) 140
3. Data Memory (data_mem) 140
4. PC Selector Module 144
J. MEM/WB REGISTER 146
K. WB STAGE 147
1. Write–Back Multiplexer 148

APPENDIX B. TEST BENCHES AND WAVEFORMS 149
A. PIPELINE REGISTERS AND VOTING LOGIC 150
1. PC Register 150
2. IF/ID Register 150
3. ID/EX Register 150
4. EX/MEM Register 153
5. MEM/WB Register 155
B. PC SELECTOR MODULE 155
C. PC REGISTER 159
D. IF STAGE 162
E. IF/ID REGISTER 165
F. ID STAGE 176
G. ID/EX REGISTER 187
H. EX STAGE 201

I. EX/MEM REGISTER 216
J. MEM STAGE 230
K. MEM/WB REGISTER 242
L. WB STAGE 247
M. UART 250
N. TEST BENCH 256

APPENDIX C. PAYLOAD PROCESSOR INSTRUCTION SET 259

LIST OF REFERENCES 261

INITIAL DISTRIBUTION LIST 265

LIST OF FIGURES

Figure 1. Standard 1U CubeSat 4
Figure 2. Eight P–PODs within a NPSCuL box..5
Figure 3. BTMR circuit structure with majority voter. 12
Figure 4. Local TMR circuit structure with one majority voter..13
Figure 5. Global TMR circuit structure with three clocks and three majority voters. 14
Figure 6. Internal TMR structure with a single clock and three majority voters. 15
Figure 7. Generic internal structure of an FPGA featuring CLBs, switching interconnects, and input/output buffers (IOBs), from [10]. 20
Figure 8. Two Actel ProASIC3 boards developed for flight testing by Parobek...........26
Figure 9. Actel ProASIC3 development board. 27
Figure 10. Digilent Genesys Virtex–5 development board..27
Figure 11. Standard layout of a Military ProASIC3EL FPGA, from [20]......................28
Figure 12. Standard VersaTile on the Actel ProASIC3EL FPGA, from [21]..................29
Figure 13. Xilinx Virtex–5 SLICEL CLB, from [24]. 33
Figure 14. Xilinx Virtex–5 SLICEM CLB, from [24]. 33
Figure 15. Standard CLB format of a Virtex–5 FPGA, from [24]..34
Figure 16. Organization of I/O banks within the Virtex–5 FPGA, from [24]..................36
Figure 17. Payload processor pipeline architecture, from [5]. 39
Figure 18. Three primary MIPS instruction formats supported by the payload processor. 40
Figure 19. Implementation of ITMR in the payload processor pipeline. 41
Figure 20. Schematic view of the payload processor IF stage modules...........................42
Figure 21. Schematic view of the payload processor ID stage modules. 43
Figure 22. Schematic view of the register file internal organization. 45
Figure 23. Schematic view of the payload processor EX stage modules. 47
Figure 24. Schematic view of the data memory module in the MEM stage. 50
Figure 25. Schematic view of the branch multiplexer and address selector modules in the MEM stage. 51
Figure 26. Schematic view of the PC selector module and its connection to the PC register..52
Figure 27. Schematic view of the WB stage multiplexer..53
Figure 28. Schematic view of the UART’s connection to the payload processor.............54
Figure 29. Waveform outputs of the triplicated PC register. 60
Figure 30. ID stage outputs for the I–format instructions ANDI, and ORI. 63
Figure 31. EX stage outputs for the I–format instructions ANDI and ORI. 64
Figure 32. WB stage outputs for the I–format instructions ANDI and ORI. 65
Figure 33. ID stage inputs and outputs for the R–format instructions AND, OR, XOR, and NOR. 66
Figure 34. EX stage outputs for the R–format instructions AND, OR, XOR, and NOR. 67
Figure 35. WB stage outputs for the R–format instructions AND, OR, XOR, and NOR. 68

Figure 36. ID stage outputs for the load word and store word instructions. 69
Figure 37. EX stage outputs for the load word and store word instructions. 70
Figure 38. MEM stage outputs for the load word and store word instructions. 71
Figure 39. ISim object panel display of the first eight bytes of data memory after simulation. 71
Figure 40. WB stage outputs for the load word and store word instructions. 72
Figure 41. ID stage outputs for the BEQ instruction. 73
Figure 42. EX stage signals while executing the BEQ instruction. 74
Figure 43. MEM stage inputs and outputs of the branch instruction. 76
Figure 44. PC selector module and PC register transitions upon receipt of a branch address. 77
Figure 45. IF stage outputs of the jump instruction. 78
Figure 46. MEM stage and PC selector module outputs of the jump instruction. 78
Figure 47. Inputs and outputs of the UART receive system during testing. 80
Figure 48. IRQ processing performed by the UART and PC selector module. 81
Figure 49. MEM stage inputs and outputs during a UART receive operation. 82
Figure 50. Load word instruction referencing the UART transmit address space. 83
Figure 51. Data memory module inputs and outputs during the UART transmission process. 84
Figure 52. UART signal transitions during a transmission. 85

LIST OF TABLES

Table 1. Minimum required I/O standards to be supported by a FPGA supporting the CubeSat payload processor 24
Table 2. I/O standards, types, voltages, and operating frequencies for the ProASIC3L, after [19]. 30
Table 3. I/O standards, types, voltages, and operating frequencies for the Virtex–5, after [25]....36
Table 4. Control module flag effects when asserted and de–asserted, after [29]. 44
Table 5. Listing of the ALU operations necessary to implement the subset of MIPS core instructions used by the payload processor. 48
Table 6. EX stage comparison flags and their assertion criteria. 48
Table 7. Gray code sequence of inputs to the pipeline registers....60
Table 8. Assembly program used for processor testing. 61

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ALU	arithmetic and logic unit
BTMR	block TMR
CLB	configurable logic block
COTS	commercial off–the–shelf
CubeSat	cube satellite
DOD	Department of Defense
ECC	error correcting code
EELV	evolved expandable launch vehicle
ESPA	EELV secondary payload adapter
EX	execution (pipeline stage)
FF	flip–flop
FPGA	field programmable gate array
GTL	Gunning transistor logic
GTLP	Gunning transistor logic plus
GTMR	global TMR
HDL	hardware description language
HSTL	high–speed transceiver logic
ID	instruction decode
IF	instruction fetch
I/O	input/output
ISR	interrupt service routine
ITMR	internal TMR
JTAG	joint test action group
LED	light–emitting diode
LET	linear energy transfer
LTMR	local TMR
LUT	look–up table
LVDS	low–voltage differential signaling
LVPECL	low–voltage positive emitter–coupled logic
MBU	multiple bit upset

MEM	memory (pipeline stage)
MIPS	multiprocessor without interlocked pipelined stages
NPS	Naval Postgraduate School
NPSCuL	NPS CubeSat launcher
PAR	place and route
PCB	printed circuit board
PCI	peripheral component interconnect
P–POD	poly–picosatellite orbital deployer
QFDR	quadruple force decide redundancy
RADHARD	radiation hardened
RAM	random–access memory
RISC	reduced instruction set computing
ROM	read–only memory
SBU	single bit upset
SEB	single–event burnout
SEE	single–event effect
SEL	single–event latch–up
SET	single–event transient
SEU	single–event upset
SDRAM	synchronous dynamic RAM
SRAM	static RAM
SSTL	stub series terminated logic
SWAP	size, weight, area, and power
TID	total ionizing dose
TIR	triplicated interwoven redundancy
TMR	triple modular redundancy
UART	universal asynchronous receiver/transmitter
USB	universal serial bus
WB	write–back

EXECUTIVE SUMMARY

The purpose of this research is to design a fault–tolerant, 32–bit, reduced instruction set computing (RISC) processor that interfaces with a universal asynchronous receiver/transmitter (UART) for a field programmable gate array (FPGA). This system serves as the first step towards developing a complete payload processor for a cube satellite (CubeSat) that will successfully interface with a sensor payload device attached via a serial connection. Development of the payload processor is a critical step in the advancement of CubeSat technology because it will provide the interface to new functionalities such as attitude control with star–trackers, imagery, and on–orbit data processing.

Outer space provides a challenging environment in which to deploy and operate electronics. A CubeSat operating in low–Earth orbit is susceptible to numerous high energy particle collisions resulting from solar wind, galactic cosmic rays, and exposure to the inner Van Allen radiation belts. Collisions between the spacecraft and these particles resulting in a disruption of electronic signals within the processor are referred to as single–event effects (SEEs). Certain types of SEEs can be prevented through the implementation of fault–tolerant logic designs. The fault–tolerant design methods considered in this thesis are capable of preventing single–event upsets, a type of SEE affecting the state of a single bit in the processor. It was determined that internal triple modular redundancy (ITMR) would be the most suitable fault–tolerant method to handle errors within the processor pipeline. This is due primarily to its ability to detect and correct single bit errors with no interruption to the processor.

The processor design features a standard five–stage pipeline with fetch, decode, execute, memory read/write, and write–back stages. Each pipeline stage is a collection of combinatorial logic components taking input from a pipeline register and providing output to the next pipeline register. The pipeline registers implement the aforementioned ITMR fault–tolerance, which is a set of three voter circuits for each data element and control signal passed between stages. The processor contains an arithmetic and logic unit capable of eleven mathematical operations. Instruction and data memory are triplicated in

the design, constituting three identical blocks of each. Both memories are byte–addressable. This was due in part to the simplicity of using the fixed–length microprocessor without interlocked pipeline stages (MIPS) instruction set architecture (ISA) and to enable the data memory to write single bytes it receives from the UART.

The UART is implemented as a memory–mapped input/output (I/O) device that writes single–byte parallel data to a specified block within data memory and reads single–byte parallel data from a specified address in data memory. An interrupt service routine (ISR) is also included within the instruction memory for data to be written by the UART. When the UART has queued data it is ready to write into processor memory, it signals the interrupt forcing the processor to save its next program counter address and jump to the ISR. Once complete, the ISR jumps back to the previously saved address and pipeline execution can continue as before the interrupt. This implementation is somewhat elementary and currently only supports one I/O interface; however, the ISR is only executed to write data from the UART to data memory and does not require context preservation of the registers.

Testing and verification of the payload processor was accomplished using behavioral simulation in two stages. First, the processor had to demonstrate it correctly handled each instruction within its ISA. This was verified by running a short program in which the processor attempts to execute at least one of each type of instruction. Using the Xilinx ISE[1] Simulator (ISim), we verified the output of each pipeline stage at each clock cycle. The second testing stage involved the integration of the UART and ISR as part of the system. Again, ISim was used to verify the processor entered and exited the ISR and wrote the data into the correct memory location. The ISA supported on the processor features 24 of the most common MIPS instructions capable of performing most functions desired on a RISC processor. Advanced multiply, divide, floating point, and other pseudo–instructions are not supported in this evolution; however, the processor can be further scaled to support their implementation.

[1] ISE® is a registered trademark of Xilinx, Inc.

A basic framework for the payload processor on which advanced payloads will eventually be supported for cube satellites was presented in this thesis. Each payload must be capable of passing data to a RISC processor that can perform processing and storage functions on its data. Future work should seek to develop a full–featured, fully tested, fault–tolerant processor supporting a wide variety of payloads. Continued development of the payload processor technology for CubeSats will offer the DOD a low–cost solution for space sensors and communications and ensure dominance in the space environment.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank God for all of the opportunities I continue to receive that enhance my career and education. I would also like to thank my family for supporting and encouraging me to work hard and finish my research, often from thousands of miles away. Prof. Herschel Loomis and Prof. Jim Newman have been fantastic advisors throughout this journey. Their patience and willingness to allow me to make mistakes was greatly appreciated and admired. Finally, I must thank the entire faculty and staff who contributed to my education while at NPS. They are a team of professionals who are truly dedicated to helping me achieve my goals, and I could not have completed my degree without them.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION AND OBJECTIVES

A. PURPOSE

Operation of electronic devices in the space environment requires increased redundancy in component design due to the increased probability of high–energy particle collisions. The sources of these particles are often the solar wind or more violent solar events such as galactic cosmic rays (GCRs), coronal mass ejections (CMEs), and solar flares. It is also possible for particles within the Earth's magnetosphere and those arriving from more distant stars to interfere with electronic devices on–orbit. Collisions with these particles can have damaging effects on spacecraft electronics. One solution to this issue is to construct electronic devices with radiation–hardened materials; however, this process raises the cost of the component significantly. Another solution is to design onboard spacecraft electronics with added logic redundancy, known as fault–tolerance. Development of fault–tolerant machines is of primary interest to the Naval Postgraduate School Cube Satellite (CubeSat) Program.

Previous research performed by the Naval Postgraduate School CubeSat Launcher (NPSCuL) program has focused on the design of a fault–tolerant launch sequencer that precisely determines the correct time and order in which to launch satellites once on orbit [1], [2]. Similar design concepts employed to create the launch sequencer can also be used to develop a payload processor, which will eventually provide an interface to a sensor device on any satellite. Methods and architectures previously used to develop the launch sequencer technology are applied towards developing a payload processor that interfaces with a serial–to–parallel I/O interface in this thesis.

Field programmable gate arrays (FPGAs) can be purchased in several different varieties, with each one offering a unique combination of logic functionality and reliability. Models generally fall within the categories of commercial, industrial, military, and radiation–hardened (RADHARD). Commercial and industrial FPGAs are suited for use in common electrical applications. Military–grade FPGAs are typically manufactured with higher quality packaging and thermal characteristics. A RADHARD model

generally refers to a FPGA that has electronic and structural features providing resistance to all types of particle events including long–term total dose effects. CubeSat research at NPS is primarily interested in the application of commercial–grade FPGAs [1], [2]. The benefits of producing a payload processor using fault–tolerant techniques embedded in a commercial off–the–shelf (COTS) FPGA are cost reduction, re–configurability, and ownership of proprietary rights to the processor design. These benefits are explained in greater detail in the following paragraphs.

Implementing fault–tolerance on a COTS FPGA is substantially cheaper than purchasing a RADHARD FPGA. A RADHARD model can cost ten to one–hundred times a COTS model. While the inherent reliability of a RADHARD FPGA is lost by using a COTS device, the possibility of more COTS devices being purchased and installed on CubeSats increases. Lowering cost and increasing reproducibility are primary motivating factors of CubeSat development. The gains made in cost and sacrifices made in hardware reliability must be met with the appropriate fault–tolerant design scheme to ensure the satellite is capable of operating in the space environment.

The primary advantage of using a re–programmable logic device, particularly an FPGA, is its ability to reconfigure and test its logic functionality throughout development and while on orbit. This introduces the possibility of resetting the processor in the event of a hard logic fault or re–defining the processor capabilities on orbit to support new missions. Partial reconfiguration of the device is also possible and would allow updates to occur while the existing configuration is still operating. The ability to reconfigure a CubeSat's processor while on orbit was discussed by Parobek in his thesis [1] but is still outside the scope of this thesis; however, establishing an initial design for a payload processor represents the next step towards implementing this capability.

Finally, by implementing a custom payload processor design on a FPGA, the Department of Defense (DOD) retains proprietary rights over the associated software files and the ability to make modifications that exclusively enhance its own mission. It might become desirable to perform specific digital signal processing (DSP) algorithms on imagery or tune an antenna to receive frequencies within a very narrow band. If the payload processor is developed from base logic components using DOD intellectual

property, these tasks can be considered as future design requirements or configuration revisions. This would provide the DOD with substantial control over the implementation of its CubeSat assets.

The goals of this thesis are as follows:

- Determine the best fault–tolerant architecture to use in the payload processor.
- Analyze two types of FPGAs and determine which device is most suitable to support the payload processor.
- Produce a software design of the fault–tolerant processor using a software package that supports schematics and HDL code.
- Interface the processor with a generic serial–to–parallel device (also designed in schematic or HDL code).
- Test the processor design using logic simulation software.

Complete design of the processor and its implementation on the selected FPGA cannot be realized at this early stage of development. A significant amount of further testing and development will be necessary to implement the processor on a FPGA. The focus of this thesis is on laying a basic foundation for the processor on which more advanced methods and technologies can be tested.

B. PREVIOUS WORK

1. NPSCuL Development

NPSCuL supports the development and testing of DOD nano–satellite technologies by deploying CubeSats as secondary payloads on DOD launch vehicles. Successful demonstrations of CubeSats with sensory payloads could result in important surveillance and communications capabilities to be gained by the DOD. CubeSats are constructed in three varieties based on their size. The smallest is a single (1U) satellite that is 100.0 mm in transverse width and 113.5 mm in height [3]. The height of the CubeSat can be elongated to make double (2U) and triple (3U) CubeSats with heights of 227.0 mm and 340.5 mm, respectively [3]. A typical 1U CubeSat similar to those developed at NPS is displayed in Figure 1.

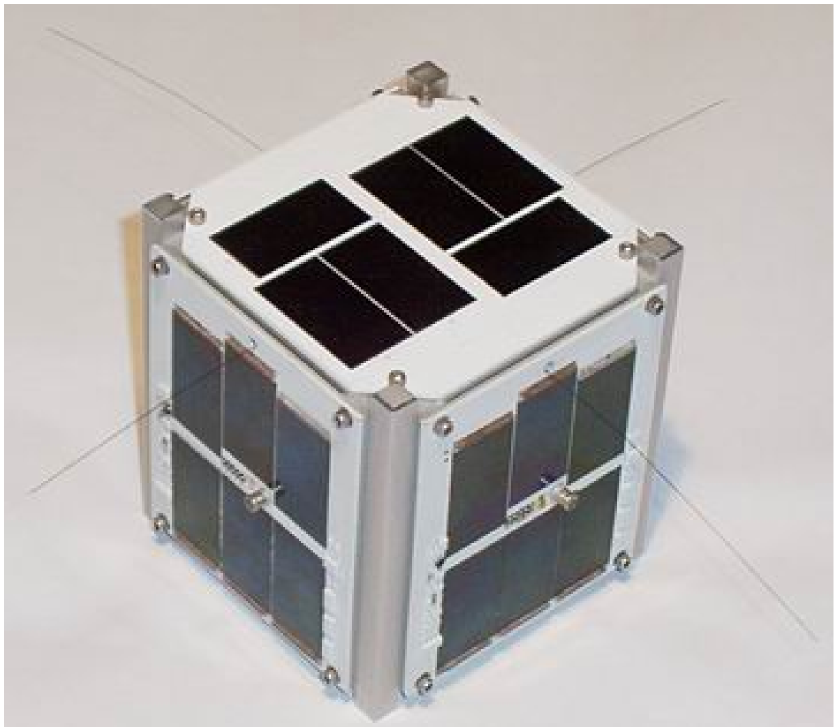


Figure 1. Standard 1U CubeSat

After a CubeSat is constructed it must be appropriately mounted on the launch vehicle for deployment. The eight modules shown inside of the NPSCuL box in Figure 2 are poly–picosatellite orbital deployers (P–PODs) and are responsible for housing CubeSats until they are ready to be deployed. The P–PODs are typically grouped up to a set of eight inside of an NPSCuL box connected to an Evolved Expandable Launch Vehicle (EELV) Secondary Payload Adapter (ESPA). The ESPA provides the mechanical coupling of the P–POD to a variety of launch vehicles. Once the appropriate orbit is achieved, the launch sequencer opens the individual P–POD doors and deploys the CubeSats, which are ejected by a spring.

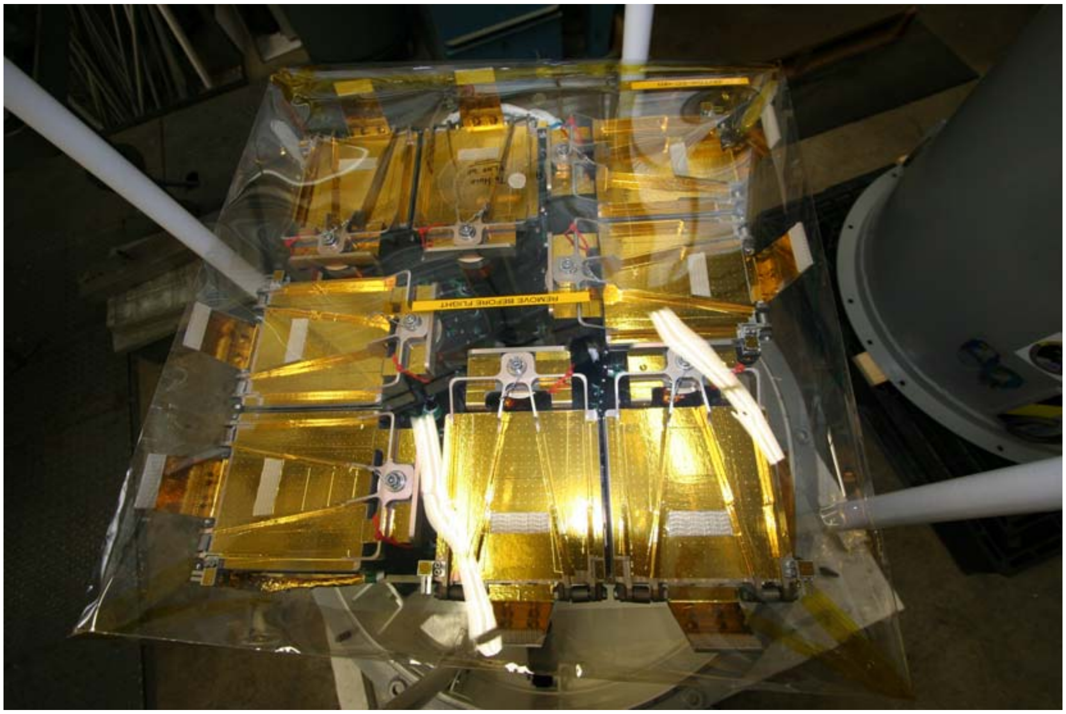


Figure 2. Eight P–PODs within a NPSCuL box.

2. Launch Sequencer

The task of reliably sequencing deployment of CubeSats from a P–POD was the motivation for the launch sequencer developed by Parobek and Brandt in their theses [1], [2]. From this work, it was determined that the Actel ProASIC3[1] FPGA was best suited to perform the launch sequencer function of the CubeSat. This was due primarily to the resistance of its flash configuration memory to single–event upsets (SEUs); however, it was noted by Parobek that for a more general payload processor, the Xilinx Virtex–5[2] model FPGA would be a better choice because of its increased computing capabilities [1]. Both products are reconsidered, and which FPGA model is most capable of implementing a payload processor is determined in this thesis.

---

1 ProASIC3® is a registered trademark of Microsemi, Inc.

2 Virtex-5® is a registered trademark of Xilinx, Inc.

3. Fault–tolerant, Pipelined Processor

Substantial influence in the design of the payload processor was generated through the thesis work of Majewicz [4]. His thesis culminated in the design of the “Pix” triple modular redundancy (TMR) processor. The “Pix” is a 32–bit, pipelined microprocessor without interlocked pipeline stages (MIPS) design that uses internal TMR (ITMR) vice an interrupt service routine (ISR) to correct SEUs encountered in its pipeline. This rapid correction of SEUs combined with the adaptability of a MIPS architecture provided a solid framework for the payload processor; however, the pre–packaged “Pix” will not automatically function as a payload processor because it does not include a method to handle I/O interrupts. It is first necessary to update the design for a new hardware description language (HDL) and version of the Xilinx ISE Webpack software suite. Additionally, an I/O interrupt handling method was implemented that determined how the processor would read and write I/O data.

C. DESIGN METHODOLOGY AND THESIS ORGANIZATION

The first step in designing the payload processor was to determine the fault–tolerance method that would best support its desired function. A consideration of several variations of both triplicated and quadruplicated logic redundancies in addition to the potential single–event effects (SEEs) is discussed in Chapter II. The ideal fault–tolerance method offers sufficient redundancy against an SEU with minimal impact to logic design resources and signal propagation time through the circuit.

A comparison of the basic performance requirements to the available resources on FPGAs, and a selection on which device and development software to implement the payload processor design is discussed in Chapter III. Generic CubeSats typically have loosely–defined design criteria. Quantification of the design parameters and requirements often is not finalized until the mission of a particular satellite is known; thus, the focus of device selection was dedicated towards determining the most capable FPGA with potential for expansion to support future missions.

Following selection of the target device, we discuss the design of the logic components and organization of the payload processor in Chapter IV. The design closely

follows a typical five–stage pipeline proposed by Patterson and Hennessy [5]. This design does not support all of the associated instructions in the MIPS core instruction set architecture (ISA). The instructions currently supported by the payload processor are detailed in Appendix C. These instructions represent a baseline set that allow the processor to perform most functions available in the core ISA. The selected instructions focus on memory loads and stores, register manipulation, and program execution.

In addition to the pipeline processor, a universal asynchronous receiver/ transmitter (UART) was constructed to serve as an input/output (I/O) interface with the processor. Generically, a UART is representative of several I/O interfaces that exist on a modern personal computer including universal serial bus (USB), RS–232C, and Ethernet. The UART employed in this thesis is not robust enough to connect the payload processor to a standardized serial interface; however, the functionality included in the control signals and data closely match the RS–232C format and were used to simulate the processor's ability to handle interrupts from an I/O device.

A summary of the testing procedures used to determine that the design was fully functional is presented in Chapter V. The first testing phase was to verify the processor could perform all instructions included in the instruction set. This was accomplished by loading the instruction memory with a short program exercising its entire ISA. This method allows troubleshooting to occur at each pipeline stage by viewing the output of that stage's combinatorial logic. Secondly, the TMR capabilities of the processor had to be tested with known errors to ensure an SEU does not propagate through more than one pipeline stage of the processor. Finally, testing of the interrupt service routine associated with the I/O interface was necessary to ensure the processor can communicate with a sensor device through a serial interface.

THIS PAGE INTENTIONALLY LEFT BLANK

II. RADIATION HARDENING AND FAULT–TOLERANCE

A. RADIATION HARDENING

In the space environment, integrated circuits are not as protected from high energy particle events and radiation as those operating below the ionosphere. Even satellites traveling in low–Earth orbit (LEO) are exposed to hazardous solar events which can induce havoc within their circuitry. The impact of long–term radiation on a computer processor is generally referred to as total dose effects and is elaborated upon in Chapter III. Immediate events caused by this radiation are referred to as SEEs; however, subdivisions of these phenomena have also been classified according to their cause and severity [6]. Particle events significant to CubeSats are detailed in the following paragraphs.

1. Single–Event Effect

SEE is the general term pertaining to damage incumbent on an electronic component resulting from a high–energy particle event. SEEs can be classified into two categories based on the type of errors they produce. Soft error SEEs cause a temporary disruption in the operation of the processor. Recovery from a soft error SEE can be accomplished by clearing the processor to a safe state before resuming operation. Hard error SEEs permanently affect the operation of one or more components in the processor. These errors cannot be overcome while on–orbit by means other than physical repair of the damaged satellite component. Hard error SEEs are rare but can cause partial loss of function or total loss of a spacecraft. The fault–tolerance methods explored in this thesis are not capable of combating hard error SEEs. FPGAs generally must be designed with additional shielding and smaller feature size among other methods to significantly reduce the probability of a hard error SEE.

2. Single–Event Upset

SEUs are soft errors that cause unintended logic signal inversions to occur in memory modules associated with the processor. SEUs can manifest as single–bit upsets

(SBUs) or multiple–bit upsets (MBUs). SBUs are the most common errors encountered in spaceflight [6] but can generally be overcome by using an error correction code (ECC) for memory read operations or fault–tolerant detection for registers. MBUs become increasingly likely to occur as the device size decreases due to the increased density of transistors [7]. This occurs because the number of transistors potentially affected by a single high–energy particle per unit area will increase. Since processor technology generally follows Moore's Law, MBUs will present more issues as devices are reduced in scale and increased in complexity. The fault–tolerant methods considered in this thesis are only guaranteed to correct SBUs, since two or three incorrect inputs to a three or four–input voter will result in an incorrect output.

3. Single–Event Transient

Single–event transients (SETs) are similar to SEUs except that they affect combinatorial logic instead of memory [6]. An error present in the combinatorial logic of the processor can propagate throughout the circuit. If a SET propagates far enough in the processor to become an input to a flip–flop (FF), it effectively becomes an SEU. SETs are dependent on the time and location of the particle strike in addition to the waveform it creates within the circuit following impact [8].

4. Single–Event Latch–up

A single–event latch–up (SEL) is a hard error that occurs when a high–energy particle causes a transistor to remain frozen in a certain state [6]. SELs generally pertain to transistors temporarily being stuck in a single state; however, SELs can progress to become a permanent single–event burnout (SEB) if corrective action is not taken. Recycling power to the device is a common method for SEL recovery [6].

5. Single–Event Burnout

A SEB is a hard error that occurs when a high–energy particle creates a short between the transistor gate and ground. SEBs may, but are not required to, result from an unattended SEL. The effects of a SEB are permanent, and the power cycling recovery

method used for SELs will not eradicate a SEB. If a CubeSat suffers a SEB, the entire mission could potentially be lost.

B. FAULT–TOLERANCE METHODS

Several fault–tolerant logic implementations are available to mitigate SEUs. Initially, techniques involving variations of TMR are explored. Then more alternative techniques including quadruple force decide redundancy (QFDR) and quadded logic are investigated. The culmination of this subsection provides resolution on the best fault–tolerance method for the payload processor.

1. Triple Modular Redundancy

TMR is accomplished by triplicating logic memory components (exclusively flip–flops and random access memory in the payload processor) and using majority voting logic to determine the correct output. Several variations of triplication are possible within a logic device. These variations typically differ by their level of granularity within the device. Finer grain TMR implementations perform voting at a lower level within the design. This implementation is typically more complex and requires more resources but provides quicker masking and correction of errors. Additionally, as transistor sizes become smaller, finer grain mitigation techniques are better suited to resist SEUs [7]. Coarser grain implementations perform logic voting at a higher level. This capitalizes on simplicity but sacrifices the localization of error handling. Four of the most common TMR implementations, block, local, global, and internal TMR, are considered for the payload processor.

a. Block TMR

Block TMR (BTMR) is implemented by using the outputs of three high–level logic devices as inputs to a voter circuit. The voter then chooses the majority of the three logic device outputs. A graphic representation of a BTMR design for the payload processor is displayed in Figure 3. The primary advantage of this approach is its simplicity. A BTMR payload processor can be realized simply by triplicating a 32–bit processor template and attaching a voter. The primary disadvantage of BTMR as applied

to the payload processor is its inability to provide reliable error correction to the system. This is because a 32–bit processor does not produce a single logic output but updates numerous registers and memory modules on each clock cycle. Additionally, one processor failure in the set results in complete loss of error correction capabilities and potentially the entire mission [7]. A more robust form of TMR is preferable to meet the high reliability requirements of the space environment.

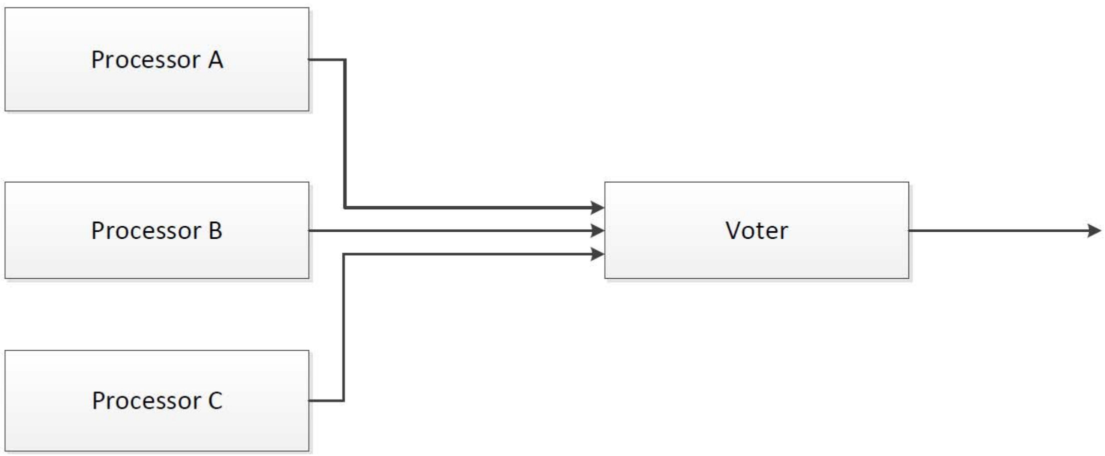


Figure 3. BTMR circuit structure with majority voter.

b. *Local TMR*

Local TMR (LTMR) implements redundancy at the flip–flop (FF) level of logic structures, as displayed in Figure 4. Each FF in the payload processor acts as part of a triple set feeding their outputs to a voter circuit. The voter circuit provides feedback to the individual FFs and a single data path to the next processing stage. LTMR has the advantage of finer grain error correction within the processor data path, making it better suited than BTMR to withstand a complete failure of error correction logic. The primary disadvantage of LTMR is its single voter, which represents a single point of failure within the circuit [7]. Error correction using a feedback path on the FF is also complex and requires the use of additional resources when implemented on the FPGA. Thus, LTMR is a more suitable solution than BTMR due to its finer grain implementation but

still contains vulnerabilities that make it too unstable to implement in the payload processor.

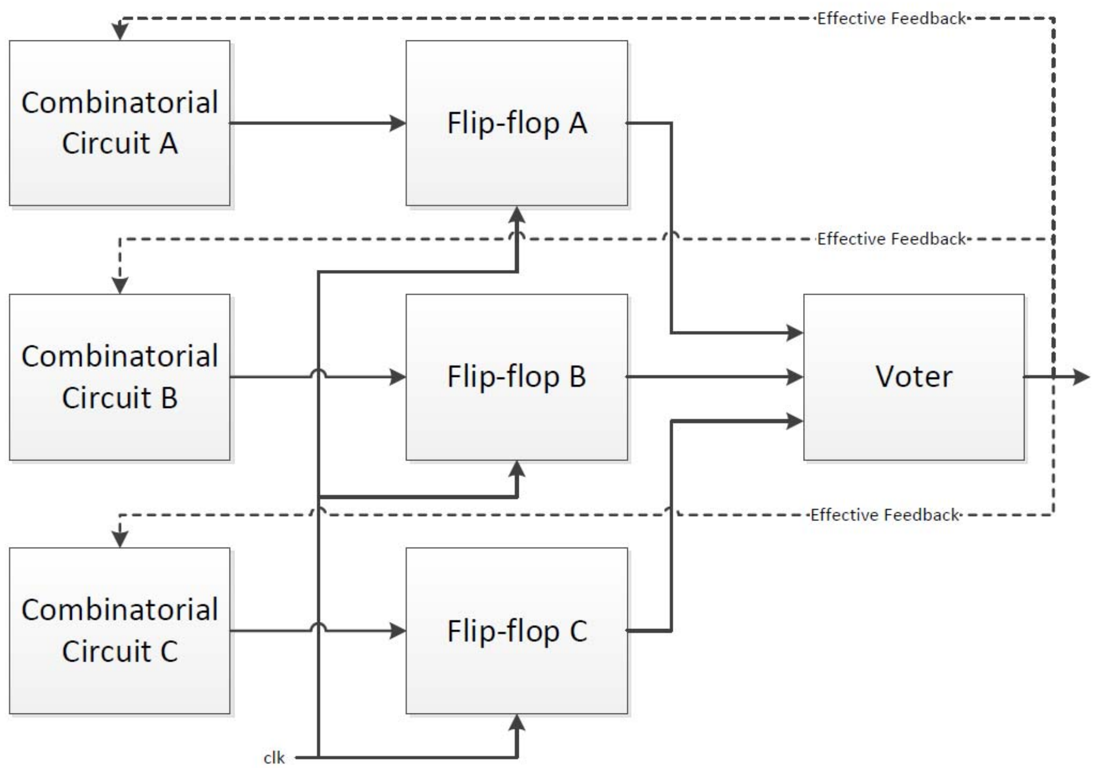


Figure 4. Local TMR circuit structure with one majority voter.

c. Global TMR

Global TMR (GTMR) effectively uses three individual processors running in parallel that output to three voting circuits, as shown in Figure 5. GTMR offers excellent resistance against SEUs since each processor operates on its own memory. Like LTMR, error correction through a feedback path is replaced by error masking through the use of three voting circuits. The greatest disadvantage of GTMR is the potential clock skew between the three processing domains [7]. GTMR also requires more logic resources, but the additional voting resources enable it to correct for SETs in a voter. The robustness of GTMR is a distinct advantage over LTMR and BTMR; however, the issue of clock skew generally outweighs the redundancy of using three individual clocks within the device

[8]. The most practical triplicated voter fault–tolerance for the payload processor is a GTMR approach with a single clock signal synchronizing all three circuits.

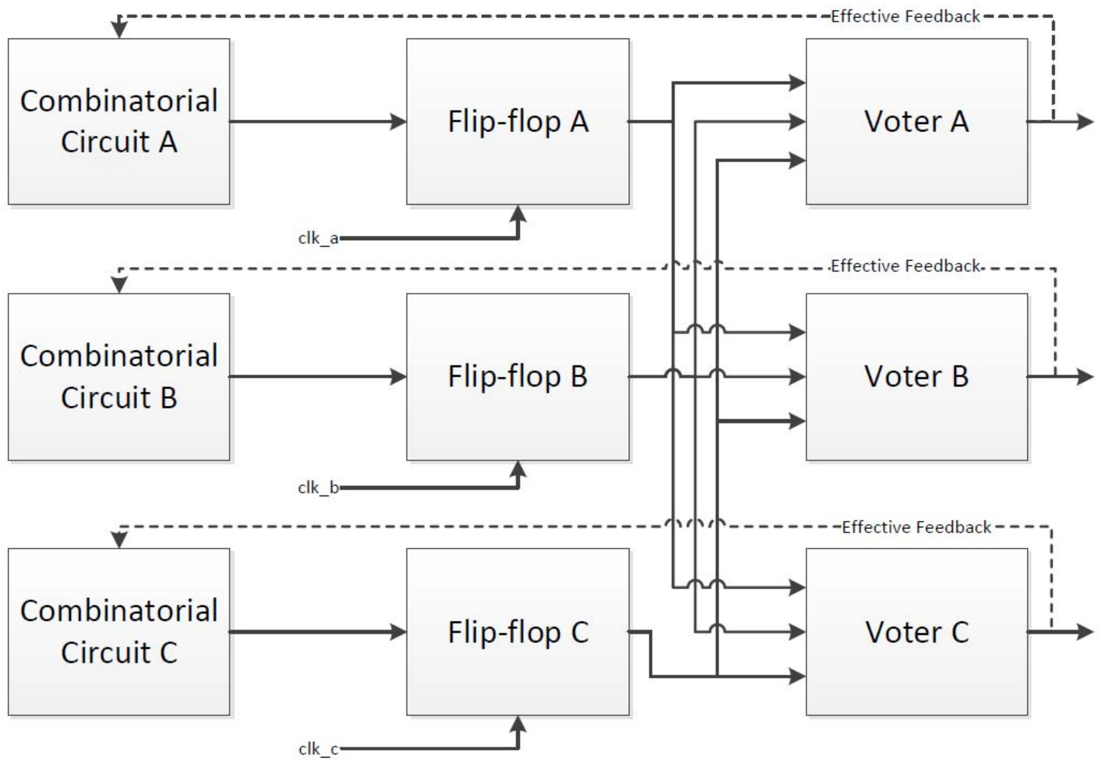


Figure 5. Global TMR circuit structure with three clocks and three majority voters.

d. Internal TMR

The concept of ITMR was proposed and implemented by Majewicz in his design of the "Pix" TMR processor [4] and is illustrated in Figure 6. It closely follows the GTMR implementation but only uses one clock for all three processors. This greatly reduces the potential for clock race conditions within the design. Like GTMR, ITMR requires additional FPGA resources for voting but effectively eliminates SEUs and SETs in the processor. Majewicz noted the vulnerability to this approach lies in the register file, where no internal voting occurs [4]; however, SEUs present in the register file are voted out after passing through a single pipeline stage. The ITMR architecture is clearly the best compromise amongst TMR variants for the payload processor.

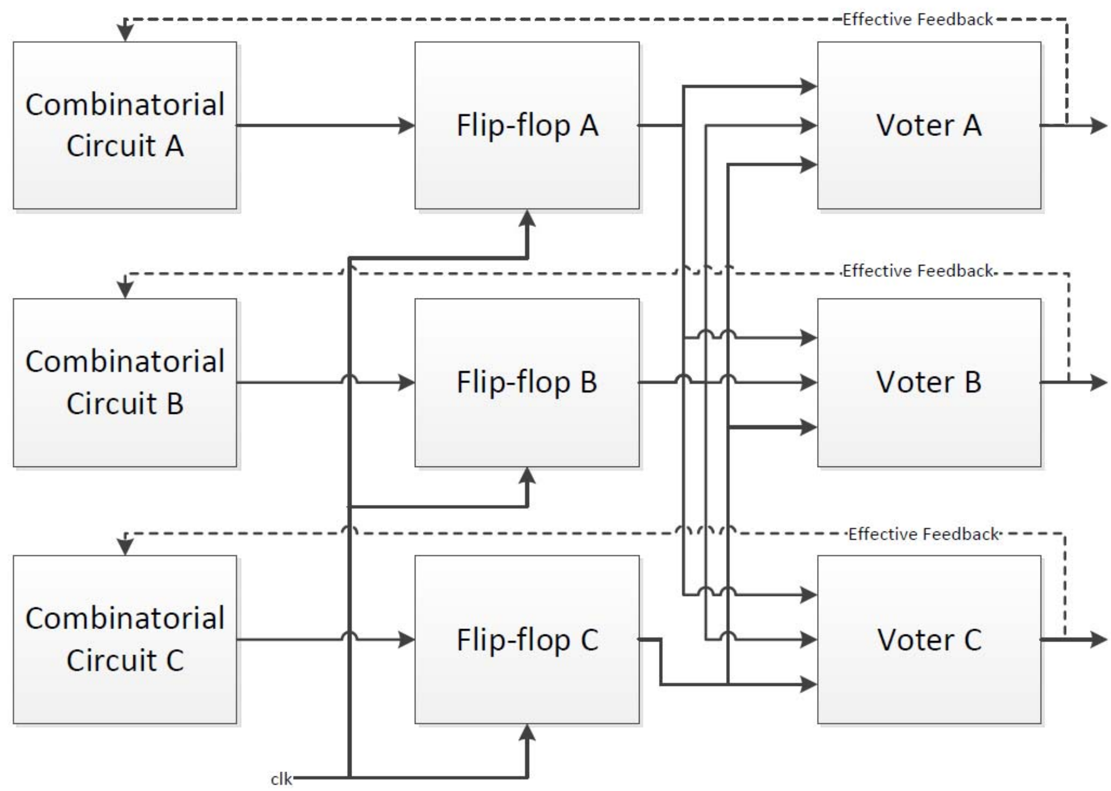


Figure 6. Internal TMR structure with a single clock and three majority voters.

2. Other Fault–tolerance Approaches

There are several other methods to implement fault–tolerance in addition to the TMR variants proposed. Parobek's research into the launch sequencer considered quadded logic, QFDR, and triplicated interwoven redundancy (TIR) [1]. Quadded logic enforces fault–tolerance by quadrupling each logic gate and its inputs and appropriately interchanging connections of the four output signals at the next sequential set of logic gates. QFDR is similar to quadded logic but decides between four inputs to a FF and look–up table (LUT) combination. This allows QFDR to be implemented at a higher logic level than quadded logic. Finally, TIR triplicates logic functions and randomly interconnects output signals between consecutive logic blocks.

Quadded logic, QFDR, and TIR do not present themselves as viable options for the payload processor. Quadded logic must be implemented at the gate level and has no built–in protection against SETs because it cannot be implemented in sequential logic.

Conversely, QFDR protects against SETs because of the presence of FFs in its design. This implementation is achievable but more complex to implement in logic design software than most of the TMR schemes considered. TIR offers no distinct advantage over traditional TMR methods and is significantly more complex to troubleshoot when designing.

3. Selection of a Fault–Tolerant Scheme for the Payload Processor

Among all the fault–tolerant schemes presented, ITMR is considered to be the best selection for the payload processor. It has a distinct advantage over the other TMR variations because it eliminates SEUs in the voter circuits as well as the combinatorial logic circuits of the processor. Majewicz noted a vulnerability of the TMR architecture in a pipelined processor was the possibility of an SEU occurring inside the register file [4]. While this temporarily increases the probability of an MBU occurring in the register file, the processor operates without issue since the voting logic ensures the error does not propagate beyond that particular pipeline stage. Ultimately, ITMR offers the best solution to eradicate SEUs in the payload processor.

C. CHAPTER SUMMARY

The motivation for fault–tolerance in space computing applications by categorizing the single–event effects that can occur while operating in the space environment was first discussed in this chapter. It was determined that the fault–tolerance method to be employed on the payload processor would provide increased resistance to SEUs, the most common SEE experienced by CubeSats. It was also noted that the selected fault–tolerance method could not provide additional protection against hard–error SEEs (i.e., SELs and SEBs), which can only be reduced by using FPGAs with better RADHARD characteristics.

Each of the candidate fault–tolerance methods was then presented for consideration. Four variations of TMR (BTMR, LTMR, GTMR, and ITMR) were investigated. ITMR was determined to be the most suitable of these schemes because of its enhanced voter logic and single clock signal. Three additional fault–tolerance schemes from Parobek’s thesis were revisited (quadded logic, QFDR, and TIR); however, ITMR

was selected for implementation in the payload processor because of its simplicity and ability to mask errors in each stage of sequential logic. Thus, ITMR was selected as the best fault–tolerance scheme to implement in the payload processor.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DESIGN REQUIREMENTS AND SELECTION OF AN FPGA DEVICE

A. INTRODUCTION

The payload processor serves as the interface between the spacecraft data bus and a sensor device the satellite uses to obtain data. It can buffer and process data received from the sensor, store data into memory, and provide data on–demand to the satellite control processor thereby allowing access to users at a ground station. The increased capabilities required of a payload processor over the launch sequencer necessitate additional performance requirements of the target FPGA. Logic cell capability, inherent radiation hardness, processor speed, memory size and configuration, and size, weight, area, and power (SWAP) are all necessary considerations for implementing a successful design.

FPGAs are organized into a large two–dimensional grid of configurable logic blocks (CLBs) interconnected by switching circuitry [9]. An example of internal FPGA structure and organization is shown in Figure 7. Each of these cells represents a certain amount of programmable logic functionality through devices such as LUTs, random access memory (RAM), logic gates, and multiplexers. The complexity and scale of the payload processor design determines the percentage of these resources required for implementation.

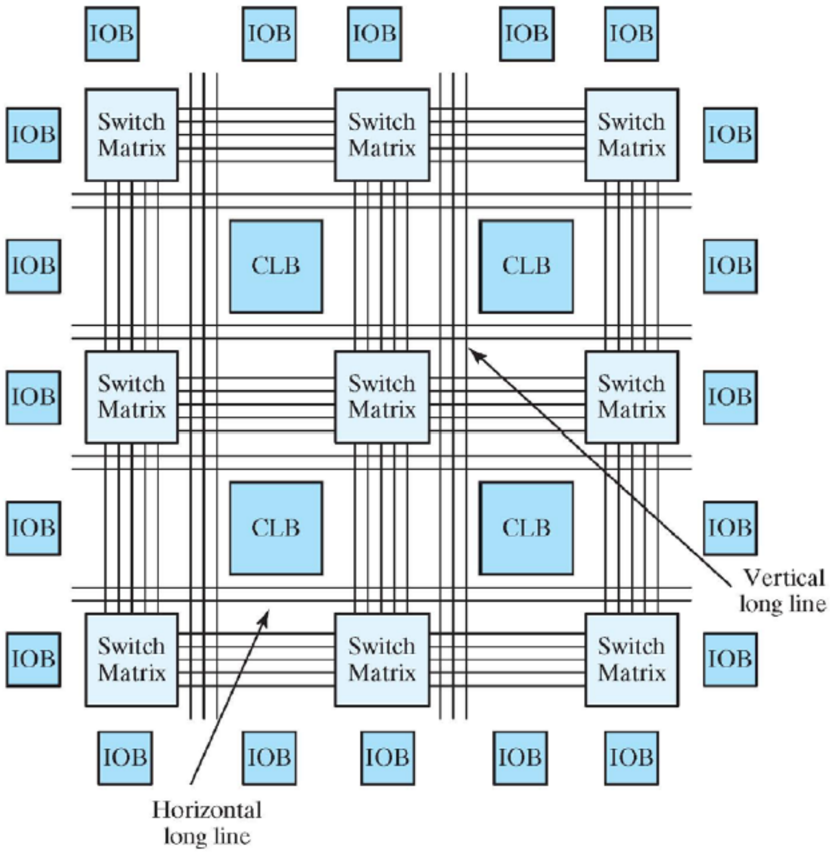


Figure 7. Generic internal structure of an FPGA featuring CLBs, switching interconnects, and input/output buffers (IOBs), from [10].

B. PERFORMANCE REQUIREMENTS

Performance requirements must be identified prior to selecting a FPGA for development. During this early phase of the payload processor technology, selection focused primarily on resource optimization vice meeting specific safety and testing requirements. These requirements become more applicable once the processor is being fitted for a specific satellite and mission. More detail on the performance requirements with regard to the payload processor concept of operations are provided in the following paragraphs.

1. CLB Capability

Each vendor offers many families and models that feature a variable quantity of CLBs providing a range of functionality. There is a continuous trade–off between the

type and quantity of these blocks with the speed, power consumption, and cost of the device. The payload processor does not impose any requirements on the capability of individual CLBs on a FPGA. Software optimizes the majority of logic functions while the place–and–route (PAR) process is being performed. Though advanced CLB capability is preferred, it is less significant compared to the macroscopic features the device has to offer.

2. RADHARD Requirements

Although device selection is limited to commercial (non–RADHARD) FPGAs, it is logical to consider any available data on a commercial FPGA's inherent radiation hardness during the selection process. Selecting a device with an above average resistance to hard error SEEs ensures the best protection against unrecoverable events the ITMR architecture cannot prevent. Each commercial model features a variety of semiconductor technologies which have variable levels of resistance to radiation. Data describing the inherent radiation hardness of non–RADHARD FPGAs from manufacturers is often unavailable. The majority of this data comes from published academic or industry research. Consequently, the methods used to obtain this data for the various FPGA models are not standardized [11]; therefore, it is necessary to gain some perspective on the characteristics that affect a FPGA's inherent radiation hardness.

In order to obtain data regarding the radiation received by a semiconductor device, it is necessary to consider the concentration and energy of incident particles. Fluence describes the number of particles passing through a unit area. Flux is defined as the amount of fluence exhibited in a period of time. A radiation absorbed dose (rad) describes the total amount of radiation absorbed by a material (i.e., semiconductor material on an FPGA). A rad is equivalent to 100 ergs per gram of energy absorbed by the material under consideration [12]. This quantity is specific to both the type of radiation under consideration and the absorbing material. Linear energy transfer (LET) expresses the amount of energy a particle transfers to the semiconductor material per unit length in units of MeV–cm^2/mg [12]. The combination of a specific LET and fluence are often used to describe the environment in which a device is operating [13]. Total ionizing

dose (TID) is a measurement of the total radiation exposure of a device during a given time period and is generally expressed in units of rad. TID is used to quantify the effects of prolonged radiation exposure a semiconductor device experiences during its lifetime. FPGAs that exhibit low LET and TID values are preferable since they are less likely to experience SEEs while on orbit.

In addition to LET and/or TID levels, transistor feature size, configuration memory type, and operating clock frequency are all relevant factors when considering which device is best suited to handle the space environment. Smaller transistors require less energy to switch and, consequently, are more susceptible to SEEs [14]. FPGAs featuring static RAM (SRAM) configuration memory are inherently more susceptible to SEEs than their flash memory counterparts [15]. Finally, increased operating clock frequency on FPGAs using FFs is associated with increased potential for SEEs [11].

There are a multitude of factors to consider regarding the radiation hardness of a particular FPGA, but device selection should not automatically favor a FPGA that possesses better inherent radiation hardness. It is also important to bear in mind that each FPGA reacts differently based on the specific model, logic design, and type of radiation incident on the device. For the payload processor, a device featuring a TID of at least 20 krad(Si) in an environment with a LET of at least 10 MeV–cm^2/mg and a fluence greater than 1.6×10^7 ions/cm^2 is sufficient for a CubeSat mission.

3. Memory Requirements

Device memory selection needs to take capacity and latency into account. Configuration memory type was accounted for as part of the RADHARD performance requirements of the FPGA. Capacity requirements of the payload processor substantially outweigh those of the launch sequencer. This is primarily due to the simplicity of the launch sequencer's design as a finite–state machine (FSM) with only 13 distinct states [2]; however, the payload processor is a complete pipelined design that runs continuously on the contents of its instruction memory and performs processing on the data it receives from its various I/O interfaces. Furthermore, the total block RAM available on the FPGA is divided into two separate memory segments for instruction and data storage. It is also

important to consider a 32–bit word (4 bytes) is the smallest usable segment of instruction memory for this processor. At a minimum, the selected device must be able to store 1 kilobyte of both instructions and data.

Memory latency was also considered as part of the selection process. This initial design of the payload processor does not maximize use of memory read and write operations; however, the purpose of the payload processor development is to eventually support more advanced sensor technologies for future evolutions of CubeSats. The possibility exists to support star trackers performing attitude control, signal monitoring antennas, and cameras to perform high–resolution imagery. These technologies frequently interact with memory and require the lowest–latency capabilities available. During this phase of development, latency was considered secondary to memory capacity and type but gains in importance as new payloads are added. Since instruction throughput is limited by instruction memory latency, the selected device should have a memory interface clocking frequency that is at least half of its maximum clock frequency. If the selected device has a memory interface clock frequency less than half of its maximum frequency, the use of caches for instruction and data memories is considered.

4. Clocking Requirements

Clock speed was considered the most critical performance requirement regarding each device’s clock management capabilities. The fastest clock available was preferred for the payload processor to reduce the time required to execute its instructions. No minimum clock speed can be established at this time since the payload processor is limited to executing a small instruction set from a small instruction memory. As payloads become more advanced, a minimum processing speed may need to be established based on the associated algorithm. A metric of interest during the design process of this thesis is clock speed when the design is placed on the device relative to the device’s maximum possible clock speed. This gives an indication of the processor’s combinatorial logic delay.

5. I/O Requirements

The payload processor must give greater consideration to the I/O capabilities of a FPGA compared to the launch sequencer. Given the size of a CubeSat, only a limited number of physical payloads can be attached prior to launch. Even the smallest payloads will likely limit a FPGA to no greater than 20 I/O interfaces. Most FPGAs easily exceed this requirement; however, the robustness of the I/O standards supported on the available interfaces a more important factor in device selection. Maximizing the number of I/O standards and placement options allows the greatest flexibility in mission planning and satellite capabilities. The selected FPGA must support the I/O standards included in left column of Table 1, which are used on some of the most common physical I/O interfaces listed in the right–hand column.

Table 1. Minimum required I/O standards to be supported by a FPGA supporting the CubeSat payload processor

I/O Standard	Commonly Associated Interfaces
Low Voltage Differential Signaling (LVDS)	SCSI, Serial ATA, PCI Express, RapidIO, FireWire, SpaceWire
Stub Series Terminated Logic (SSTL)	DDR SDRAM, PCI Express
Peripheral Component Interconnect (PCI)	USB/Serial, Disk Drives, Network Cards

6. SWAP Requirements

All spacecraft are subject to limitations on SWAP resources. Power was the primary SWAP performance requirement considered for this thesis since the payload processor aims to greatly expand its computing capabilities over the launch sequencer. There is no strict requirement on the power ratings of batteries or other power sources carried onboard a CubeSat [3]. Ideally, the payload processor should consume a small fraction of the power used by its associated payload. FPGA manufacturers do not often publish power consumption explicitly because it is highly dependent on the number and type of I/O applications being used among other factors. The devices considered in Section C of this chapter are evaluated based on the maximum and normal operating voltage and current characteristics instead.

Size, area, and weight limitations are primarily dictated by the type of satellite. The FPGA and associated hardware components are required to fit on a printed circuit board (PCB) to be placed within the CubeSat rails. The interior length and width dimensions of a CubeSat are not directly specified by [3]; however, scale models in the NPS CubeSat laboratory suggest a PCB length and width of approximately 80 mm fits well within the CubeSat rails. The weight of the FPGA is relatively small even compared to the CubeSat. Once development reaches a phase where a complete PCB design and associated payload are required, size and weight become significant factors.

C. CANDIDATE FPGA DEVICES

Two candidate devices for initial implementation and testing of the payload processor are considered in this thesis. The first of these devices was the Actel (now Microsemi, Inc.) ProASIC3 model FPGA, which had been previously selected for implementation of the launch sequencer. Parobek chose this device primarily because its flash–based memory was more resistant to SEUs and SELs, and it provides a memory state immediately after power was applied [1]. The ProASIC3 logic architecture, though small, was sufficient to support the launch sequencer state machine. The second device considered was the Xilinx Virtex–5 model FPGA. This device is significantly more powerful in processor and I/O capabilities than the ProASIC3 but was considered a less qualified device for the launch sequencer because of its SRAM configuration memory. The utility of these devices in the context of a complete payload processor must be reconsidered given new design criteria. This analysis should not serve as the final selection of a device for space–flight but rather to help determine the type and quantity of future applications the payload processor is able to support.

Multiple variants of each FPGA family are available on the market. The ProASIC3 family features the ProASIC3E, ProASIC3 nano, ProASIC3L, and radiation tolerant ProASIC3 variants. Xilinx manufactures the generic Virtex–5, Virtex–5Q (defense–grade), and Virtex–5QV (space–grade) devices [16]. The generic Virtex–5 features the LX (high–performance general logic), LXT (high–performance logic with advanced serial connectivity), SXT (high–performance signal processing with advanced

serial connectivity), TXT (high–performance systems with double density advanced serial connectivity), and FXT (high–performance embedded systems with advanced serial connectivity) [17]. Further elaboration of both the ProASIC3 and Virtex–5 devices is made in the following sections with respect to the performance requirements discussed in Section B of this chapter.

As of this writing, the NPS CubeSat laboratory possesses four FPGA development boards. Three of these boards feature ProASIC3 family devices, while only one features the Virtex–5. Two of the ProASIC3 boards are identical and were custom developed as flight test models by Parobek [1]. These boards are depicted in Figure 8 and feature the "A3PN250" model of the ProASIC3 family, eight light–emitting diodes (LEDs), power adapter, and joint test action group (JTAG) chain. The third ProASIC3 development board, shown in Figure 9, was a development board product purchased from Actel. It contains the A3P1000 model FPGA, eight LEDs, and eight switches. The Virtex–5 is featured in the Digilent Genesys development board, displayed in Figure 10. This board features eight LEDs, eight switches, three buttons, display screen, video output, USB–A/B, RS–232C, Ethernet, JTAG, and mini–USB ports. The mini–USB ports are used exclusively to program the device directly via the Xilinx iMPACT or Digilent Adept software programs.

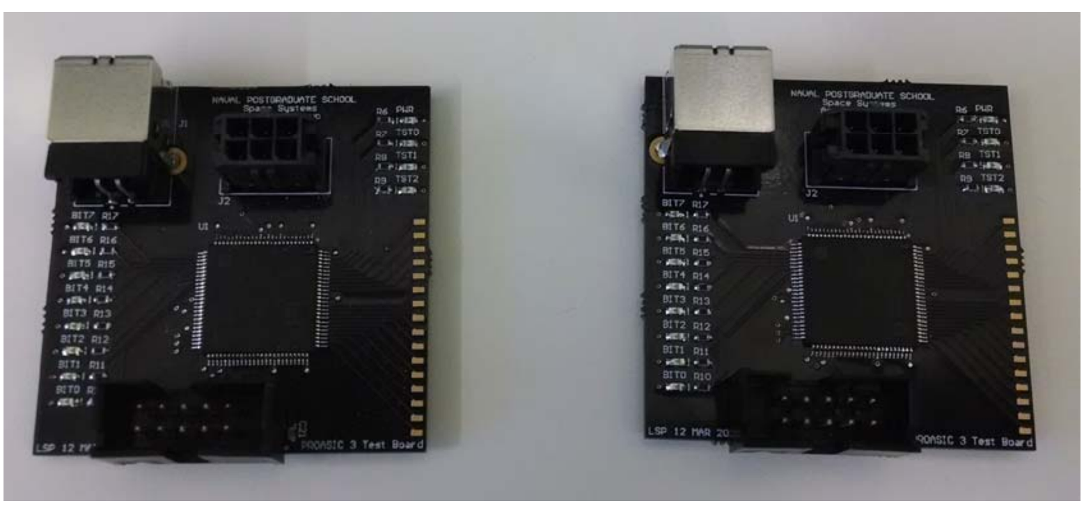


Figure 8. Two Actel ProASIC3 boards developed for flight testing by Parobek.

Figure 9. Actel ProASIC3 development board.

Figure 10. Digilent Genesys Virtex–5 development board.

1. Actel ProASIC3

The pertinent performance characteristics for the Actel ProASIC3 family of devices are summarized in the following sections.

a. Architecture

The ProASIC3 FPGA is optimized for cost and minimum power consumption while maintaining competitive processing speed [18]. A physical layout of the ProASIC3EL device is displayed in Figure 11. The hardware architecture is accomplished through the use of proprietary CLBs called VersaTiles. The ProASIC3 FPGA contains 384 – 24,576 VersaTiles depending on the device model [19]. The standard layout of a VersaTile is depicted in Figure 12. Each VersaTile accepts four input signals and can represent one of four logic design functions:

- A three–input logic function
- Latch with clear or set
- D–flip–flop with clear or set
- Enable D–flip–flop with clear or set

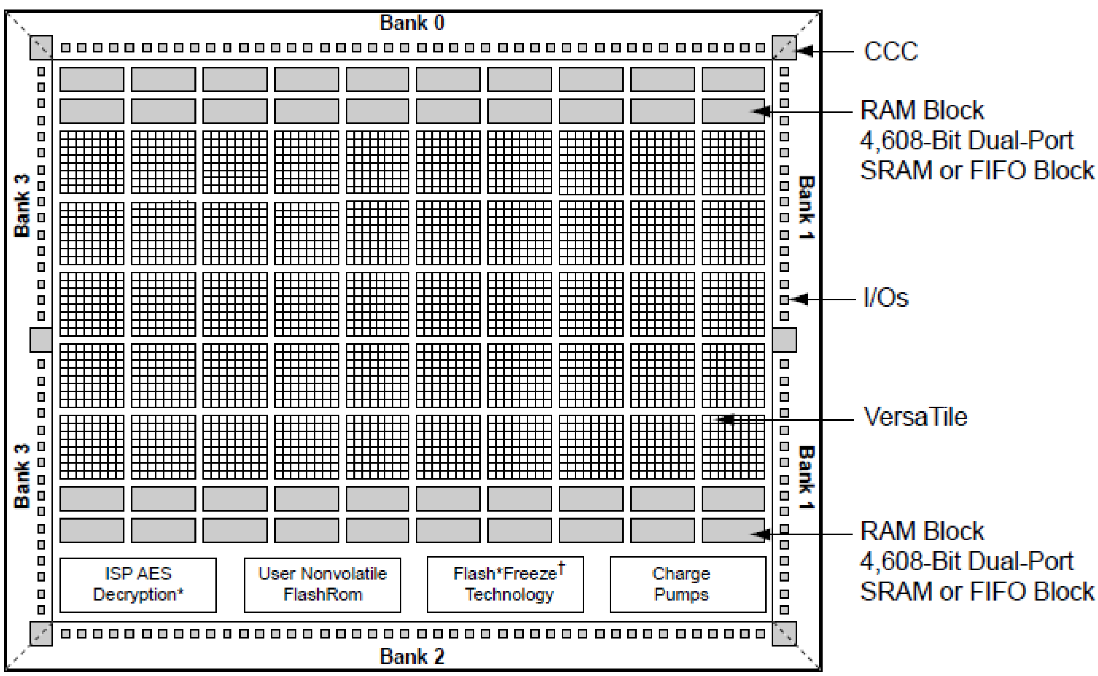


Figure 11. Standard layout of a Military ProASIC3EL FPGA, from [20].

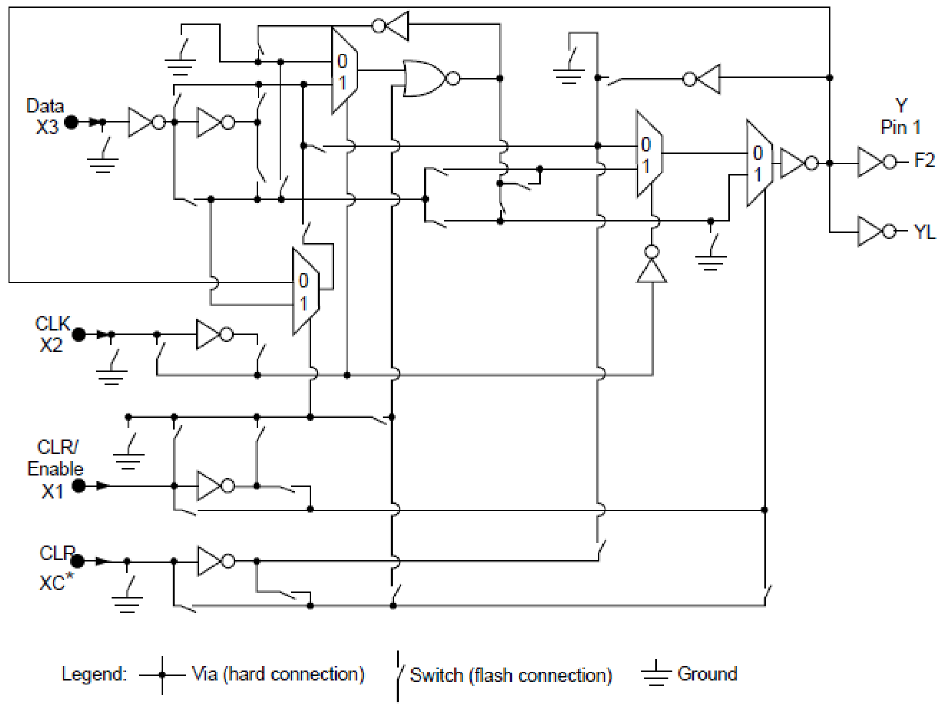


Figure 12. Standard VersaTile on the Actel ProASIC3EL FPGA, from [21].

b. Radiation Tolerance

Radiation testing performed by Poivey et al. on the ProASIC3 A3P3000L model produced data describing the response of the device to heavy ion and proton induced SEEs for multiple shift register, memory, and clock configurations [22]. It was determined the ProASIC3 model under consideration can resist the occurrence of SELs and configuration memory SEUs from a LET of at least 55 MeV–cm^2/mg and 1x10^7 ions/cm^2 of fluence [22]. The TID of the device tested was 65 krad(Si). Signal propagation times through combinatorial logic increased by 50% throughout the FPGA as the device approached its TID [22]. As expected, the flash–based ProASIC3 exhibited excellent TID tolerance and resistance to hard–error SEEs during performance testing.

c. Memory Capabilities

The ProASIC3 family of FPGAs features both a configuration flash memory and SRAM data memory on the device. The flash configuration memory functions as a read–only memory (ROM) and maintains its contents after power is no longer applied to the

device. The data memory contained in the ProASIC3 family ranges from 36,864 – 516,096 bits of SRAM divided into blocks of 4,608 bits each. This capacity translates to at most 16,128 32–bit words that can be stored in SRAM. This memory operates only in synchronous mode, using separate read and write clocks operating at any desired frequency up to 250 MHz. Multiple configurations of width and depth size can be specified via the designated write and read width pins.

d. I/O Capabilities

The ProASIC3 family of FPGAs supports two, four, or eight I/O banks depending on the model selected, ultimately enabling the device to support 68–620 I/O interfaces [19]. Each of the I/O banks contain several mini–banks that provide 8–18 individual I/O pins [19]. Each of the I/O devices on a single mini–bank share a common reference voltage designated by configuring one I/O pin as the controller for that mini–bank [19]. General I/O banks on the ProASIC3 can support differential voltages of 1.2 V, 1.5 V, 1.8 V, 2.5 V, and 3.3 V [19]. The robust "mini–bank" structure and array of supported voltages allow the ProASIC3 FPGAs to support the I/O standards displayed in Table 2.

Table 2. I/O standards, types, voltages, and operating frequencies for the ProASIC3L, after [19].

Standard	Type	Voltage(s) (V)	Frequency (MHz)
LVTTL	Single–Ended	3.3	< 200
LVCMOS	Single–Ended	1.5/1.8/2.5/3.3	< 200
PCI	Single–Ended	3.3	33/66
PCI–X	Single–Ended	3.3	33/66/133
HSTL (Class I and II)	Voltage Referenced (0.75 V)	1.5	< 400
SSTL (Class I and II)	Voltage Referenced (1.25 V/1.5 V)	2.5/3.3	Not Listed
GTL	Voltage Referenced (0.8 V)	2.5/3.3	20 – 40
GTLP	Voltage Referenced (1.0 V)	2.5/3.3	20 – 40
LVPECL	Differential (+/– 850 mV)	3.3	Not Listed
LVDS	Differential (+/– 350 mV)	2.5	< 200

e. Clock Management

The ProASIC3 family FPGAs features a total of six global clocks available for routing on across the chip [20]. An additional four clock signals are available for each quadrant in the device [20]. These signals are controlled by clock conditioning circuitry, each of which contains a phase–locked loop (PLL). In the normal power operating mode (V_{cc} = 1.5 V), each clock frequency can be set between 0.75 – 350 MHz with a duty cycle between 48.5 – 51.5% [19]. The low power operating mode (V_{cc} = 1.2 V) on the ProASIC3EL can produce clock signals between 0.75 – 250 MHz with a duty cycle between 48.5 – 51.5% [23].

f. Power Consumption and Distribution

Each device in the ProASIC3 family supports a 1.5 V mode of operation [19]. The ProASIC3EL variant also features a 1.2 V mode of operation [23]. The recommended operating ranges for core input voltage are 1.425 – 1.575 V for the 1.5 V normal operating mode [19] and 1.14 – 1.575 V for the 1.2 V mode on the ProASIC3EL [23]. Each device will handle maximum voltage limits from –0.3 – 1.65 V for short periods.

ProASIC3 devices achieve the goal of delivering low–power computing capabilities through two features of the device. First, power–on and configuration loading is accomplished using non–volatile flash memory. Since flash memory stores charge to retain its state, it is available immediately following power–on [21]. SRAM initializes to an indeterminate state which can cause “inrush” currents to draw additional power during the power–on stage [21].

The ProASIC3 FPGA’s second power management feature is the low power modes that can be implemented once power–on and normal operating conditions are achieved. There are four low power modes able to be implemented on the ProASIC3 model FPGA [21]:

- Static (Idle) Mode
- User Low Static (Idle) Mode
- Sleep Mode
- Shutdown Mode

Both static mode and user low static mode cease clock function but retain the current states of all SRAM, I/Os, and registers [21]. These devices draw minimum current to maintain their state while in either mode [21]. The ProASIC3EL features two additional modes within the static mode of operation known as flash freeze modes, which retain the states of memory devices like static mode but do not stop clocking [21]. Sleep Mode shuts off voltage to the entire FPGA core and requires logic states to be saved to nonvolatile memory in order to restart in the same state [21].

2. Xilinx Virtex–5

The pertinent performance characteristics for the Xilinx Virtex–5 family of devices are summarized in the following sections. The information summarized focuses primarily on the VLX50T variant found on the Digilent Genesys development board. For information regarding other models and features, the reader should refer to [17], [24], and [25].

a. Architecture

The Xilinx Virtex–5 FPGA features a two–dimensional array of CLBs, each containing two sub–elements called logic slices. There are two types of logic slices in the Virtex–5: SLICEL and SLICEM. Each SLICEL element can be programmed to represent one of four standard logic functions [24]:

- Look–up Tables
- Storage Elements
- Wide–function Multiplexers
- Carry Logic

The SLICEM element implements these same four logic functions but also contain 256 bits of distributed RAM and a 128–bit shift register [24]. The SLICEL CLB is displayed in Figure 13 and the SLICEM CLB in Figure 14.

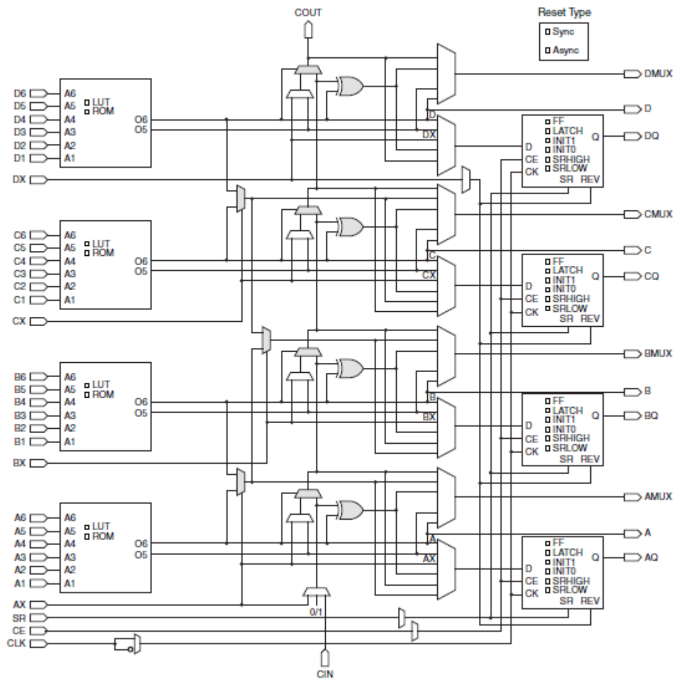


Figure 13. Xilinx Virtex–5 SLICEL CLB, from [24].

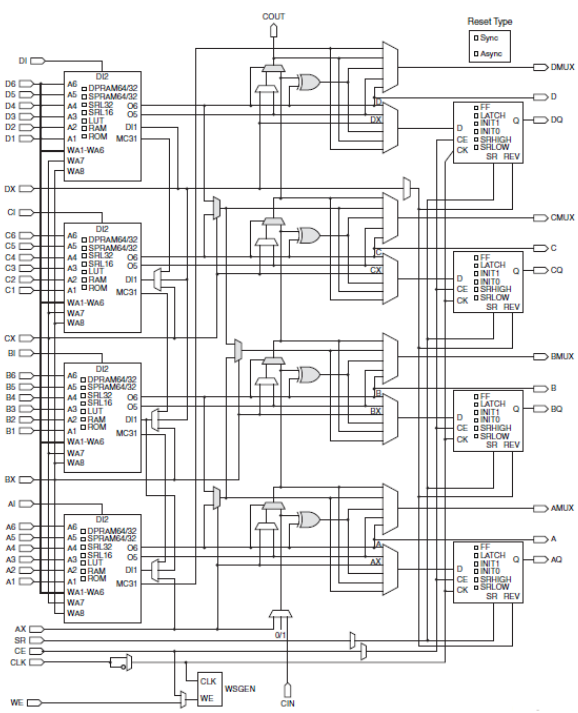


Figure 14. Xilinx Virtex–5 SLICEM CLB, from [24].

The row and column arrangement of CLBs and their associated slices is displayed in Figure 15. Each CLB is connected to a switching matrix, which is further connected to the general routing matrix of the FPGA [24]. SLICEM CLBs are featured in alternating columns moving across the FPGA [24]. A total of 7,200 logic slices are available in the VLX50T model, 1,800 of which are SLICEM variants [17].

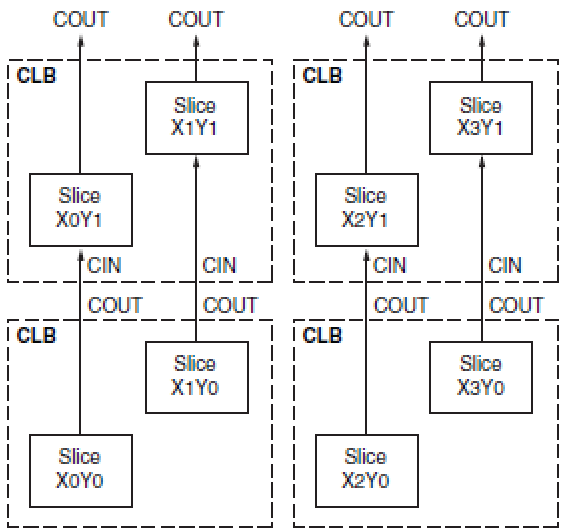


Figure 15. Standard CLB format of a Virtex–5 FPGA, from [24].

b. Radiation Tolerance

Radiation testing for the Virtex–5 was performed by Hiemstra et al. to determine the characterization of SEUs on an SRAM based FPGA [26]. This experiment was performed on the same FPGA model featured on the Genesys development board in the NPS CubeSat laboratory. The Virtex–5 was tested to an equivalent of 53 years of heavy ion fluence using a LET of 10 MeV–cm^2/mg [26]. This resulted in a TID of 23.8 krad(Si) exhibiting no SELs or other hard–error effects [26]. The length and intensity of the Virtex–5 testing is noticeably shorter than the tests performed on the ProASIC3 [22]; however, the Virtex–5 performed well during these tests and satisfied the desired performance criteria for radiation hardness.

c. Memory Capabilities

The Virtex–5 features SRAM configuration memory, which loses its memory state when power is removed from the device. Data storage is implemented using 36 kbit SRAM blocks that can support a multitude of word size and depth combinations. The total number of memory blocks available depends on the specific device chosen. The VLX50T variant features 60, 36 kbit RAM blocks, resulting in 2.16 Mbits of total memory [24]; however, other models within the Virtex–5 family are capable of supporting up to 18.5 Mbits of block RAM [17]. The capacity of the VLX50T translates to 67,500 total words that can be stored in block RAM on the specified FPGA. The Virtex–5 block RAM operates at the same frequency (450 – 550 MHz depending on the speed grade) as the primary clock for the device [25].

d. I/O Capabilities

The VLX50T device features 15 I/O banks supporting 480 user I/O interfaces [17]. Organization of the I/O banks is depicted in Figure 16 for a VLX30 device. The I/O banks each typically contain 40 I/O inputs for the outer columns [24]. The organization of the inner column of the FPGA varies based on the total number of banks and interfaces featured on the device [24]. A reference voltage pin is automatically allocated for one of every 20 pins if a single–ended I/O standard requiring a differential input buffer is used [24]. Each low–voltage I/O standard implemented on the device must use an I/O bank for its associated output drive voltage [24]. The I/O standards supported by the Virtex–5 are listed in Table 3.

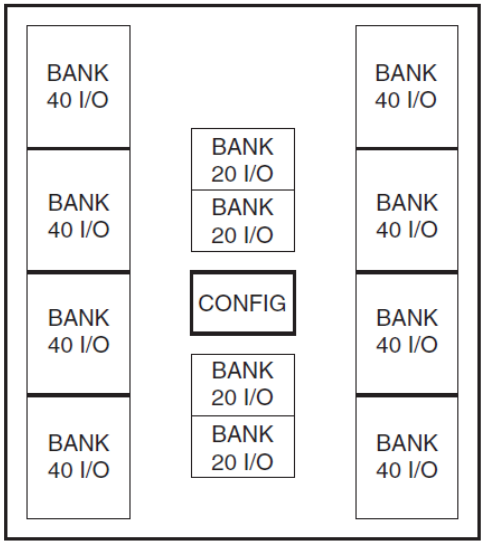


Figure 16. Organization of I/O banks within the Virtex–5 FPGA, from [24].

Table 3. I/O standards, types, voltages, and operating frequencies for the Virtex–5, after [25].

Standard	Type	Voltage(s) (V)	Frequency (MHz)
LVTTL	Single–Ended	3.3	< 200
LVCMOS	Single–Ended	1.2/1.5/1.8/2.5/3.3	< 200
PCI	Single–Ended	3.3	33/66
PCI–X	Single–Ended	3.3	66/133
HSTL (Class I/II/III/IV)	Voltage Referenced (0.75 V)	1.5/1.8	< 400
HSTL (Class I)	Voltage Referenced ()	1.2	Not Listed
SSTL (Class I and II)	Voltage Referenced (1.25 V)	1.8/2.5	Not Listed
GTL	Voltage Referenced (0.8 V)	2.5/3.3	20 – 40
GTLP	Voltage Referenced (1.0 V)	2.5/3.3	20 – 40
LVPECL	Differential (+/– 850 mV)	3.3	Not Listed
LVDS	Differential (+/– 350 mV)	2.5	< 200
Differential HSTL (Class I/II)	Differential	1.5/1.8	Not Listed
Differential SSTL (Class I/II)	Differential	1.8/2.5	Not Listed
RSDS	Differential	2.5	Not Listed

e. Clock Management

The Virtex–5 is capable of implementing up to 32 global clock signals [24]. Each global clock signal is generated through a global clock buffer [24]. On the physical FPGA, these clock buffer components are driven by six clock management tiles (CMTs) built into the FPGA [24]. The Virtex–5 datasheet does not offer a single clock speed for reference like the ProASIC3 [25]. Rather, it lists numerous clock speeds based on the speed grade and applicable logic construct being used. Numerous combinatorial and sequential logic modules are necessary to create the payload processor; however, most 32–bit components can be implemented at a clock speed of 450 MHz for the speed grade of the VLX50T available at the NPS CubeSat laboratory [25]. Higher speed grades in the Virtex–5 family feature clock speeds up to 550 MHz for many 32–bit logic structures [25].

f. Power Consumption and Distribution

The typical operating mode of the VLX50T model FPGA requires a supply voltage between 0.95 – 1.05 V and 2 mA of current [25]. Increased speed grades and processing rates for various Virtex–5 models require more current [25]. The internal core cannot handle voltages outside the range of –0.5 – 1.1 V, and no greater than +/–100 mA of current may flow through any pin on the device [25]. The Virtex–5 does not feature any special modes comparable to those of the ProASIC3 family to minimize power consumption.

D. FPGA SELECTION

The two FPGAs considered are representative of the variety of reconfigurable devices manufactured to meet the needs of several different industries. The ProASIC3 device represents a low–power, low–cost solution with impressive inherent RADHARD qualities. The Virtex–5 has substantially more logic resources, a faster clock, and supports more I/O standards than the ProASIC3. It is worth noting each of the FPGAs met the initial performance requirements; therefore, it is possible the payload processor can potentially be implemented and tested on either model. It is the author's view that the Virtex–5 should be the initial development platform for the payload processor, primarily

because it offers the greatest potential for expansion as the technology continues to develop.

E. CHAPTER SUMMARY

The performance requirements for a FPGA that can implement the payload processor were first discussed in this chapter. The categories of CLB capability, memory, I/O capability, clock management, and SWAP each played some role in the selection of a device. The performance characteristics of two FPGAs considered potential candidates to support the processor were then summarized. It was determined the Virtex–5 would serve as a better model for development of the payload processor because its logic resources, clock speed, and I/O standards offered greater potential for future expansion of the technology.

IV. DESIGN OF THE PAYLOAD PROCESSOR

A. PIPELINED PROCESSOR COMPONENTS AND ORGANIZATION

The payload processor was designed as a standard five–stage pipeline using a subset of the MIPS core ISA proposed by Patterson and Hennessy [5], shown in Figure 17. This processor features instruction fetch (IF), instruction decode (ID), execution (EX), memory (MEM), and write–back (WB) stages, each separated by a pipeline register. The payload processor is effectively three of these processors running in parallel on a common clock, with ITMR voter circuits at each of the inputs to the triplicated pipeline registers. The goal of the ITMR implementation is to provide sufficient redundancy to correct SEUs encountered at any individual stage logic or any pipeline register at the stage input.

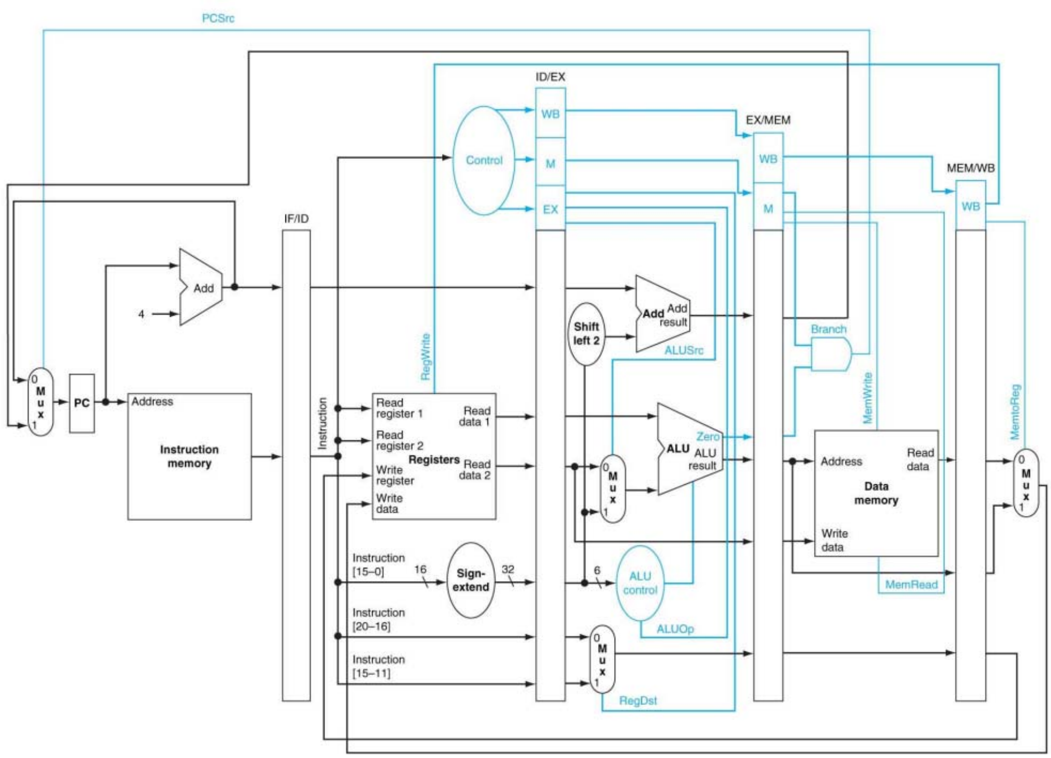


Figure 17. Payload processor pipeline architecture, from [5].

The standard MIPS instruction formats executed by the processor are shown in Figure 18. The MIPS ISA features three primary instruction variants: I–format, R–format, and jump format. Every MIPS instruction is 32 bits in length. The six most significant bits always contain an opcode which provides a more specific identification of the function. I–format instructions take two register arguments and a 16–bit offset. Generally, the remaining register (**rt**) is used as a destination for the result of the operation. However, branch I–format instructions compare the **rs** and **rt** registers and determine a new address using the 16–bit offset. The R–format instruction takes two register addresses (**rs** and **rt**) and places the result of the ALU operation in a third register (**rd**).

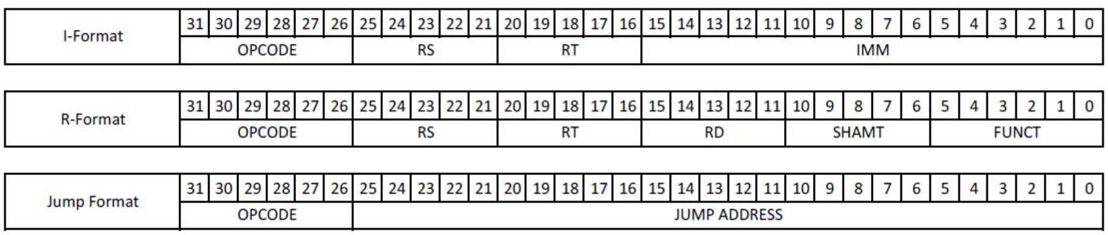


Figure 18. Three primary MIPS instruction formats supported by the payload processor.

B. PAYLOAD PROCESSOR DESIGN

The standard pipeline design and ITMR architecture had to be meshed into a single design for the payload processor. A conceptual structure for the payload processor is displayed in Figure 19. The pipeline stages feature combinatorial logic and reside on either side of a pipeline register. Each of the triplicated processors passes their outputs to a set of registers, where the values are held until the next positive edge of the clock signal. During the next positive clock edge, each of the three sets of registers pass their values to three voters. The voters produce the same result because they all have the same inputs. This ensures each of the triplicated processors has the majority voted signal during the next stage of combinatorial logic, and any SEUs have effectively been eradicated.

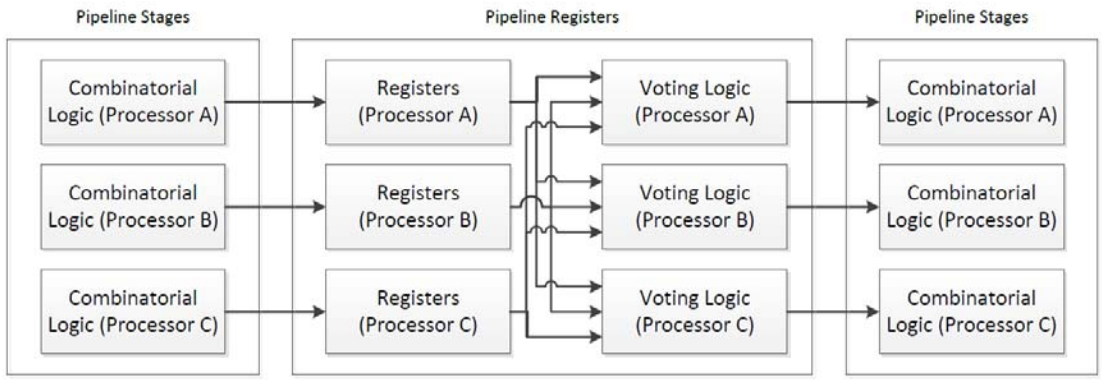


Figure 19. Implementation of ITMR in the payload processor pipeline.

Numerous logic components were necessary to build a pipelined processor. The combinatorial logic components were grouped into five pipeline stages. Each pipeline stage took inputs from and passed outputs to a pipeline register. This process begins with the program counter (PC) register passing an instruction address to the IF stage. The instruction address points to an instruction located in memory, and the instruction data is then passed to the ID stage. The instruction is decoded in the ID stage, and the register operands are obtained. The EX phase performs mathematical operations using the ALU and calculates a possible branch address. The MEM stage can then read or write to data memory and determine if a branch or jump from the next PC address is appropriate. Finally, the WB stage writes new data from memory or the ALU back to the register file. Each of these stages and their associated logic components are discussed in greater detail in the following sections.

1. IF Stage

During the IF stage, a PC address is translated into an actual MIPS instruction. The instruction memory acts as a ROM and forwards the instruction located at the index specified by the PC to the IF/ID register. Additionally, the PC plus four bytes address must be calculated and sent back to the PC selector module. Since the payload processor does not yet implement advanced features such as data forwarding and hazard detection, the programmer must ensure at least four no–operation (NOP) instructions follow a

branch or jump instruction. This prevents the processor from operating on instructions prior to the feedback address of a branch or jump instruction being received at the PC.

The schematic organization of the IF stage is displayed in Figure 20. The input signals **pc_a**, **pc_b**, and **pc_c** are the triplicated PC addresses. These inputs are sent to the **inst_mem_trip** module which houses the byte–addressable instruction memory and **npc_addr_adder_trip** module which calculates the PC plus four address. The output of the **inst_mem_trip** module is then forwarded to the ID stage. The jump address is also calculated during the IF stage. Per standard MIPS jump instruction format, the jump address is a concatenation of the four most significant bits of the input PC address, the 26 least significant bits of the jump instruction, and two zeroes [28]. The latter ensures the jump address references the first byte of a word–length instruction. This address is then forwarded through each of the pipeline stages and pipeline registers until a jump feedback signal is generated in the MEM stage.

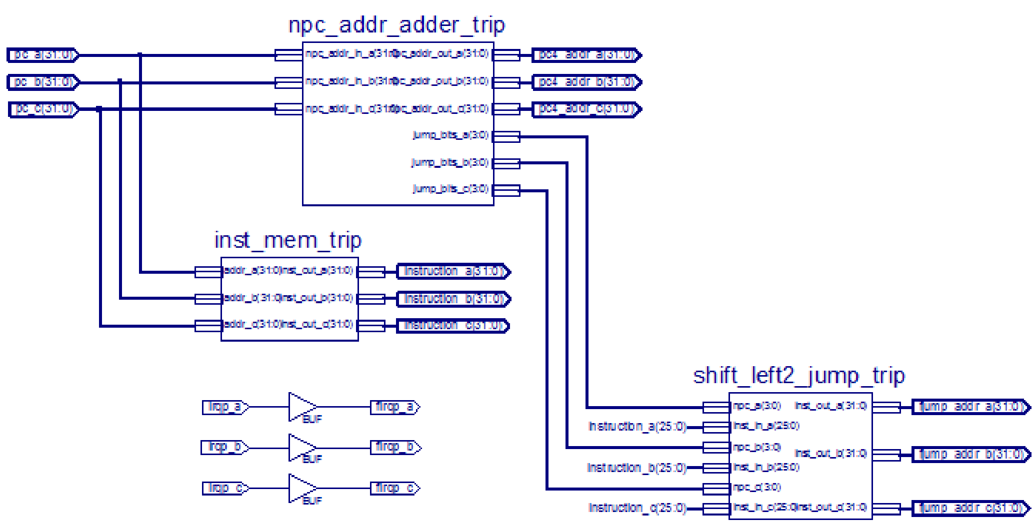


Figure 20. Schematic view of the payload processor IF stage modules.

2. ID Stage

The purpose of the ID stage is to decode an instruction into its various subcomponents and trigger the appropriate control signal flags. A schematic layout of the ID stage is displayed in Figure 21. The first major component featured in the ID stage is

the **control_tmr3** module, which contains three control modules that use the **opcode**, **funct**, and **rt** register fields of an instruction to determine the appropriate ALU operation code and flag settings. The effect of each of these individual flag settings are described in Table 4.

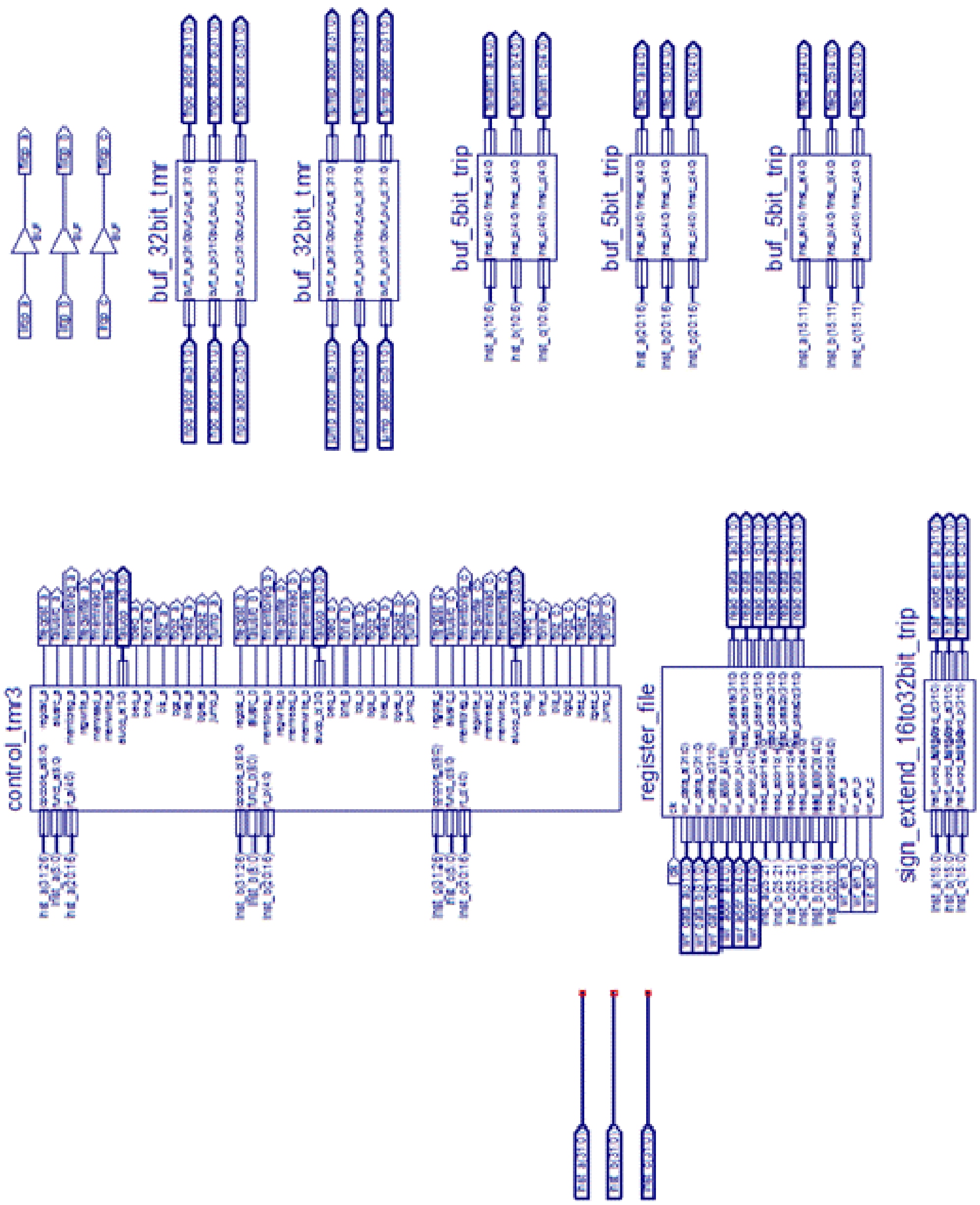


Figure 21. Schematic view of the payload processor ID stage modules.

Table 4. Control module flag effects when asserted and de–asserted, after [29].

Signal	Effect when De–asserted	Effect when Asserted
regdst	The write–back register address is contained within the rt register field (inst. bits 20 – 16).	The write–back register address is contained within the rd register field (instruction bits 15 – 11).
regwrite	None	The write–back register specified by the instruction will be written with new data from the ALU or data memory during the WB stage.
alusrc	The second ALU operand comes from the second register file output.	The second ALU operand comes from the immediate field (lowest order 16 bits of the instruction)
memread	None	Instruction is a load word. Signal is passed as a read enable to data memory. This triggers the memory to read a specified word to write–back to the register file.
memwrite	None	Instruction is a store word. Signal is passed as a write enable to data memory. This triggers the memory to write a word from the register file.
memtoreg	The value used to write–back to register comes from ALU.	The value used to write–back to the register comes from data memory.
beq	None	Triggers the next PC address to be the calculated branch address if the ALU inputs are equal. Used only with the beq instruction.
bne	None	Triggers the next PC address to be the calculated branch address if the ALU inputs are not equal. Used only with the bne instruction.
blz	None	Triggers the next PC address to be the calculated branch address if the first ALU input is less than zero. Used only with the blz instruction.
bgz	None	Triggers the next PC address to be the calculated branch address if the first ALU input is greater than zero. Used only with the bgz instruction.
blez	None	Triggers the next PC address to be the calculated branch address if the first ALU input is less than or equal to zero. Used only with the blez instruction.
bgez	None	Triggers the next PC address to be the calculated branch address if the first ALU input is greater than or equal to zero. Used only with the bgez instruction.

A second major component in the ID stage is the **register_file** module, which holds each of the 32, 32–bit registers used by the processor. A more detailed schematic view of the register file's internal organization is displayed in Figure 22. The register file is unique amongst all combinatorial logic components featured in the processor in that it must use a clock to perform its operation. During the design process, we determined the register file should be capable of reading from two registers and writing to one register on each clock cycle. This eliminates one additional clock cycle delay that is otherwise required to update data during the WB stage.

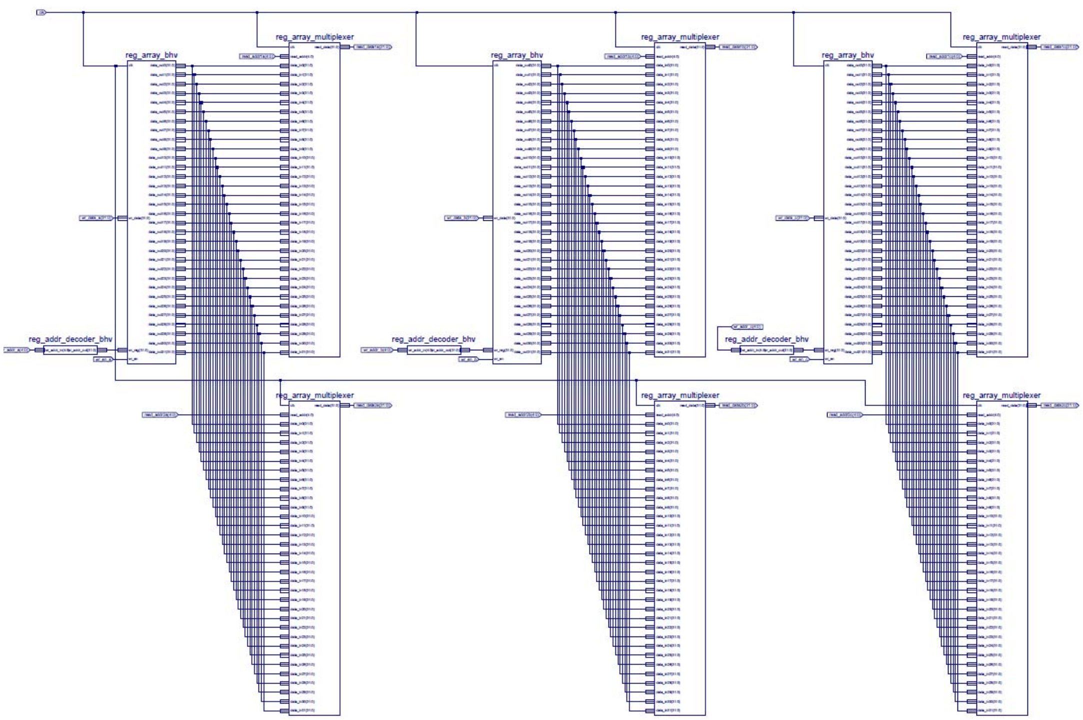


Figure 22. Schematic view of the register file internal organization.

Read and write operations within the register file each operate on different logic components. The register file receives three parallel addresses and data inputs to write to registers. The write register address is expressed as a 5–bit value and is decoded to one of 32 individual write enable signal by the **reg_addr_decoder_bhv** module. These signals are then distributed to the various registers along with the write data, but only one register receives the write enable signal per clock cycle. This register rewrites its old data with the

write input data while the clock cycle is high. When the clock signal is low, two sets of three parallel read address inputs are forwarded to one of six **reg_array_multiplexer** modules. These modules read each of the 32 register states (including the register which was just written on the previous half of the clock cycle) and select the appropriate register as an output based on the decoded input read address. The three **reg_array_multiplexer** modules in the upper–half of Figure 22 operate in parallel for the first register operand input to the ALU. The modules in the lower–half operate in parallel for the second register operand input to the ALU.

3. EX Stage

The EX stage performs mathematical and logical operations on data extracted from the register file and immediate (IMM) field of the instruction. The results of these operations are forwarded to the MEM and WB stages for storage in data memory or the register file, respectively. A schematic layout of the EX stage components is shown in Figure 23. The EX stage heavily relies on the **addr_adder_trip** and **ALU_trip** modules to accomplish its functions. The EX stage performs branch address calculation using the **addr_adder_trip** module. It calculates the branch address by adding the offset specified in the IMM field to the PC plus four address calculated in the IF stage. This result is forwarded to a PC selector module and onto the PC register if a branch is taken.

The **ALU_trip** module performs all of the necessary mathematical functions required to implement the processor's ISA. The first of two operands is received from the register file. The second ALU operand is received from the **multiplex_2to1_nbit_trip** module which selects between the second value forwarded from the register file and the IMM field. The selector bit for the **multiplex_2to1_nbit_trip** module is the **alusrc** flag. Eleven ALU operations are required to implement the payload processor. These operations and their corresponding ALU codes are displayed in Table 5. Many other mathematical and logical operations are possible in a MIPS architecture, but a simple ISA was desired for this initial iteration of the payload processor.

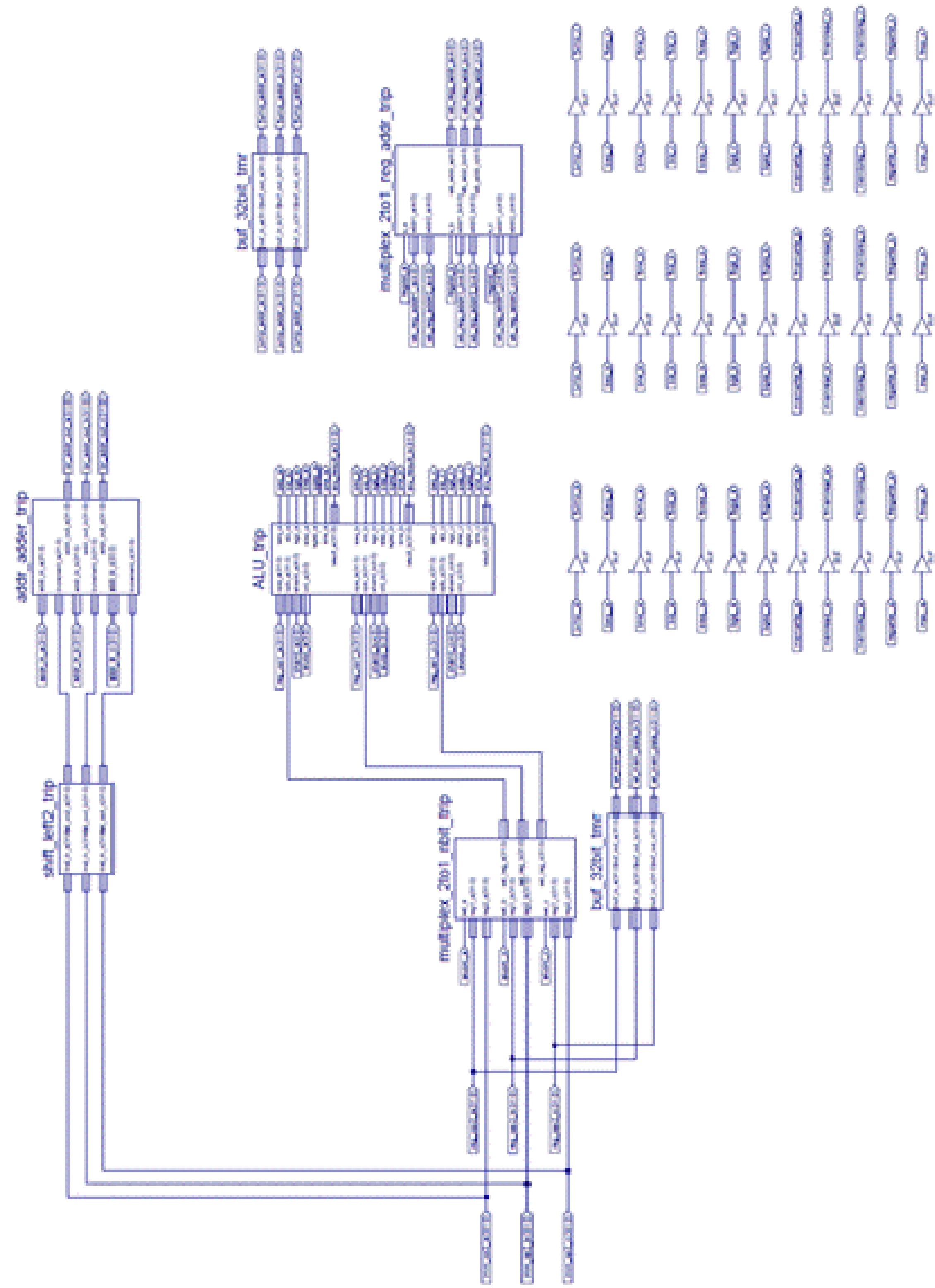


Figure 23. Schematic view of the payload processor EX stage modules.

Table 5. Listing of the ALU operations necessary to implement the subset of MIPS core instructions used by the payload processor.

Control Code	Operation
0	Addition
1	Subtraction
2	Logical AND
3	Logical OR
4	Logical XOR
5	Logical NOR
6	Shift Left Logical
7	Shift Left Variable
8	Shift Left Logical
9	Shift Left Variable
10	Set on Less Than

In addition to the mathematical operations featured in Table 5, six comparison flags are set for each pair of inputs to the ALU. These comparison flags and their assertion criteria are summarized in Table 6 and match one–to–one with the branch flags generated in the ID stage. These comparison bits and branch flags are evaluated during the MEM stage to determine if the next PC address should be the resulting calculation of the **addr_adder_trip** module.

Table 6. EX stage comparison flags and their assertion criteria.

Comparison Flag	Assertion
seq	Both ALU operands are equal
sne	The ALU operands are not equal.
slz	The first ALU operand is less than zero.
sgz	The first ALU operand is greater than zero.
slez	The first ALU operand is less than or equal to zero.
sgez	The first ALU operand is greater than or equal to zero.

4. MEM Stage

The MEM stage serves two primary functions in the payload processor. First, it contains the triplicated data memory where words can be loaded into or stored from the register file. Secondly, it is where the processor determines if the conditions for a branch or jump instruction have been met and what the next appropriate address is for the PC

register. A detailed explanation of the implementation of these functions is the focus of this section.

The payload processor data memory module is contained within the **data_mem_trip** component, displayed in Figure 24. This memory is divided into three equal segments of 45 kilobytes (kB) each. The memory is byte–addressable, but only one word may be read or written during each clock cycle. This is because the load word and store word instructions are currently the only instructions that can manipulate data memory. Each of these operations requires a complete clock cycle per pipeline stage.

The data memory component of the MEM stage can accept inputs from and read outputs to either the register file or UART. The functionality of the UART read and write operations are explained in Section C of this chapter, but perform a very similar operation on a separate address space in the memory. For standard read operations, the address fields **addr_a**, **addr_b**, and **addr_c** receive the forwarded output of the **ALU_trip** module which represents a data memory address. Likewise, during a store word instruction, the ID stage forwards data from the register specified in the **rt** field of the instruction. Both read and write operations must be accompanied by read and write enable signals which are linked to the memread and memwrite flags generated in the ID stage. The **ALU_trip** module result is also forwarded past the data memory module to the WB stage via the **buf_32bit_tmr** module for instructions other than load or store which writes the result to the register file.

Determination of a branch or jump instruction is performed by the **branch_multiplexer_trip** and **addr_sel_module_trip** modules displayed in Figure 25. The overall output of these modules is forwarded to **pc_select_module** which makes the final selection of the next PC address by taking pending interrupt requests into account. The **branch_multiplexer_trip** module receives the branch flags generated in the ID stage and the comparison bits from the EX stage as inputs. If a branch flag and its corresponding comparison flag are both set, the conditions for a branch instruction have been met, and the **branch_jump_sel_out** output pin of the multiplexer is set high. Next, the branch address calculated in the EX stage and jump address calculated in the IF stage are received as inputs to the address selector module along with the branch and jump

flags at the address selector module. If a branch or jump flag is set high, the module selects the corresponding address to forward back to the PC selector module and sets the **br_jump_sel_out** family of outputs to logic high. If neither flag is set high, the module does not forward a new address or set the **br_jump_sel_out** outputs high. This results in the PC selector module choosing a different PC address on the following clock cycle.

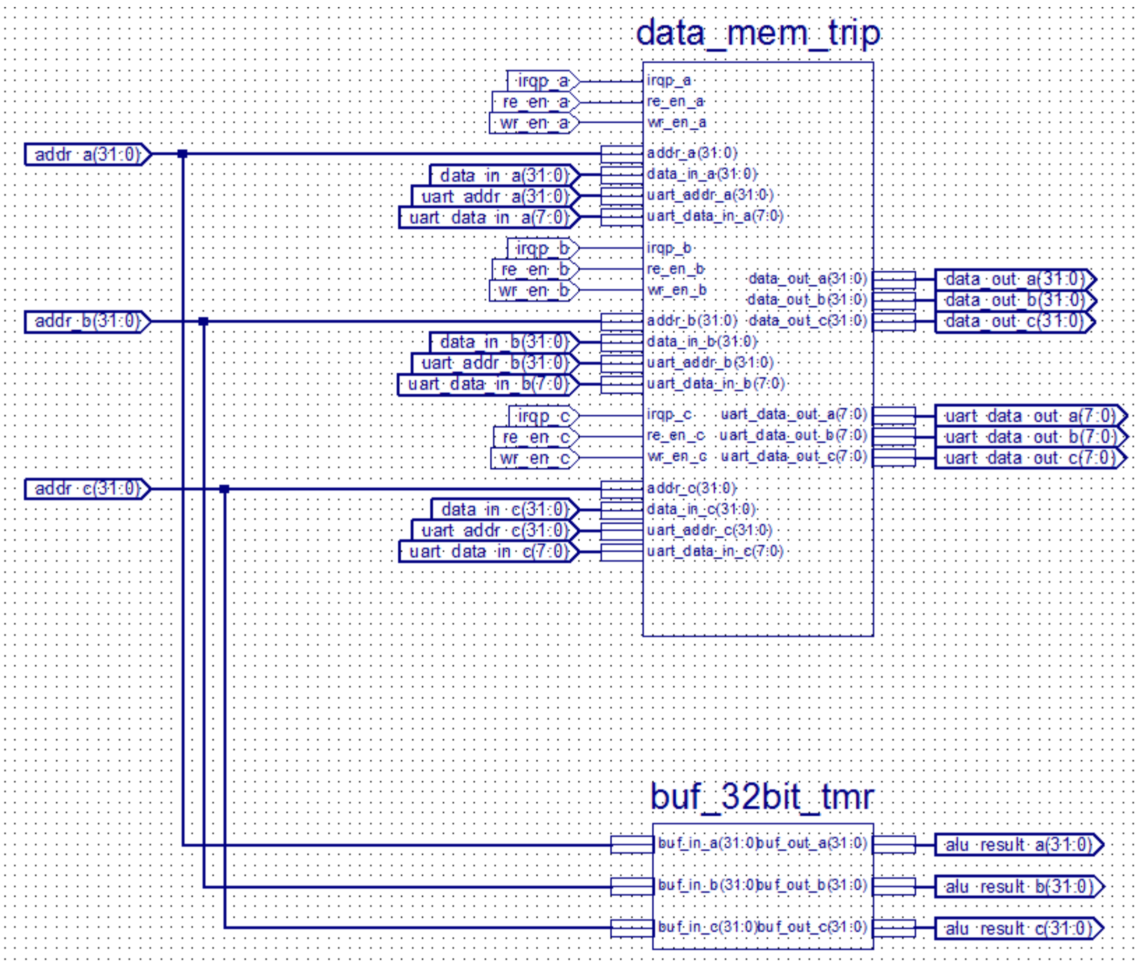


Figure 24. Schematic view of the data memory module in the MEM stage.

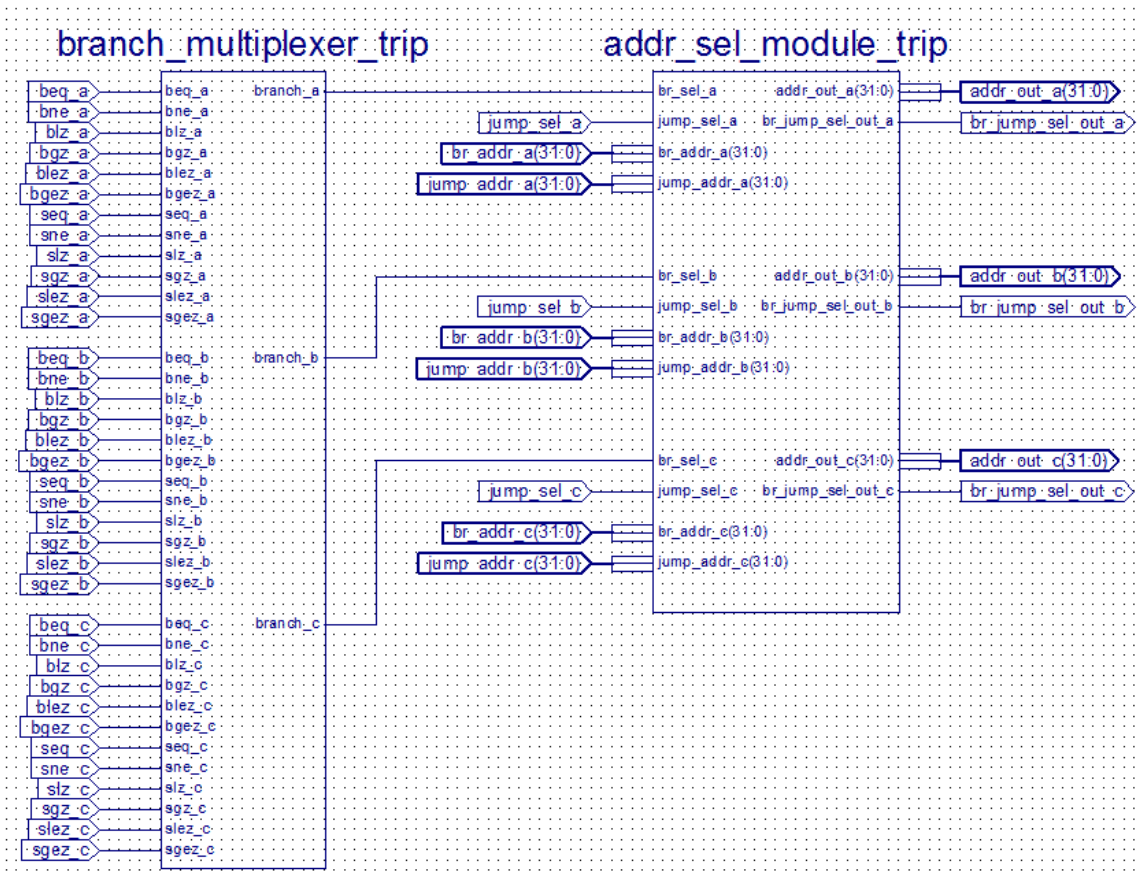


Figure 25. Schematic view of the branch multiplexer and address selector modules in the MEM stage.

After the branch multiplexer and address selector have completed execution, the result is forwarded to the **pc_select_module_trip** component, displayed along with the PC register in Figure 26. The purpose of the **pc_select_module_trip** is to choose whether the next PC address is that of the current PC plus four, the forwarded branch or jump address, or the start of the ISR if an interrupt is present. The module gives highest priority to an interrupt request (IRQ). If an IRQ from the UART is present, it forces the processor to execute the ISR. This mode is discussed in greater detail in Section C of this chapter. If an IRQ is not present but the **br_jump_sel_out** flag is set high, the module uses the forwarded branch or jump address as the next PC. Finally, if neither an IRQ nor the branch/jump flag is present, the module selects the PC plus four address calculated at the IF stage. Refer to Subsection 1 of this section. The PC plus four address present at **pc_select_module_trip** following a branch is not the PC address of the instruction

immediately following a branch instruction. Four nop instructions must be inserted after a branch to ensure instructions are not executed unintentionally while the processor is making a determination on the branch criteria.

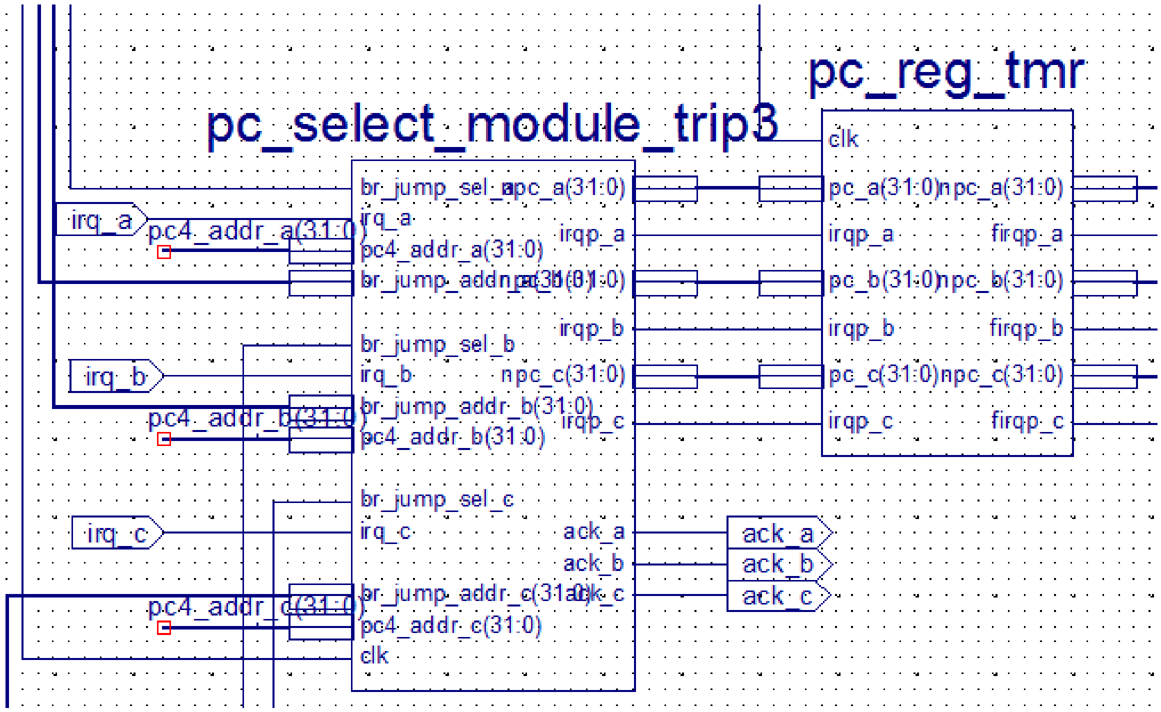


Figure 26. Schematic view of the PC selector module and its connection to the PC register.

5. WB Stage

The WB stage essentially marks the final step in the execution of an instruction, and consists of only one multiplexer, displayed in Figure 27. Every instruction in the payload processor's ISA, with the exception of store word and jump, writes back new data to the register file. This write–back data comes from either the data memory module (if the instruction is a load word) or the ALU result. The selector bit for this multiplexer is the memtoreg flag generated in the ID stage. The triplicated ALU result is connected to pins **input1_a**, **input1_b**, and **input1_c** in Figure 27 while the triplicated data memory output is connected to the **input2_a**, **input2_b**, and **input2_c** pins. Per Table 4, if the **memtoreg** flag is set low, the multiplexer forwards the ALU result into the register file. If the **memtoreg** flag is set high, the multiplexer forwards the data memory output

instead. The new data and register address reach the register file and rewrite the register during the same upper–half of the clock cycle.

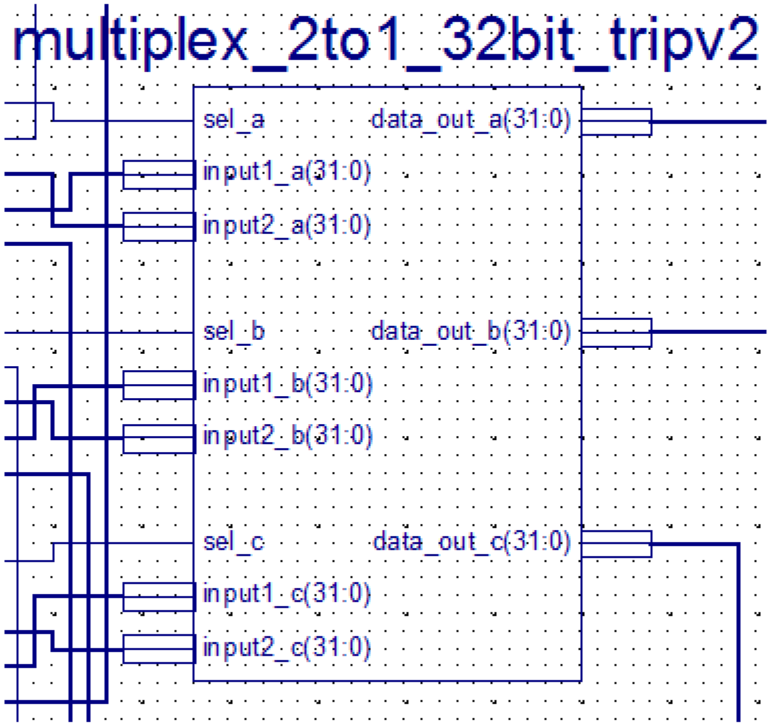


Figure 27. Schematic view of the WB stage multiplexer.

C. UART DESIGN

1. Concept

The UART is a generic serial–to–parallel interface the processor pipeline uses to perform data transactions with an I/O device. A single UART module is currently the only I/O interface included on the payload processor as shown in Figure 28. Consequently, it alone generates the IRQ to the processor when it receives data to transfer to memory. This action is completed using an interrupt–driven I/O scheme which calls an ISR built into instruction memory and requires no action by the programmer. The ISR first clears the pipeline, allowing instructions in progress to update the appropriate registers. It then stores the received byte into a data memory address that has been mapped to the UART. This address is stored as data in addresses 22496 – 22499 of data memory. It can be read from memory and tracked by the programmer to determine if new

data has been written to the UART memory space. As additional bytes are written to data memory, the current write address is incremented and replaced in its reserved memory space. Once the UART writes to address 44995 (the end of UART memory space), the address wraps around to address 22500 (the beginning of UART memory space) where it begins overwriting old UART data with new receptions.

The UART also transmits byte–wide parallel data on a serial interface with another I/O device by calling a load word function referencing memory that is mapped to the UART interface. The programmer performs this action by using a store word instruction to write to a data memory address allocated for UART transmission. A load word instruction referencing this memory address then triggers data memory to forward the byte at the same address to the UART. No specific protocol is currently supported by the UART; however, the UART code adapted from [30] was oriented towards a RS–232C interface. The UART's internal operation and interaction with the processor pipeline are described in the following paragraphs.

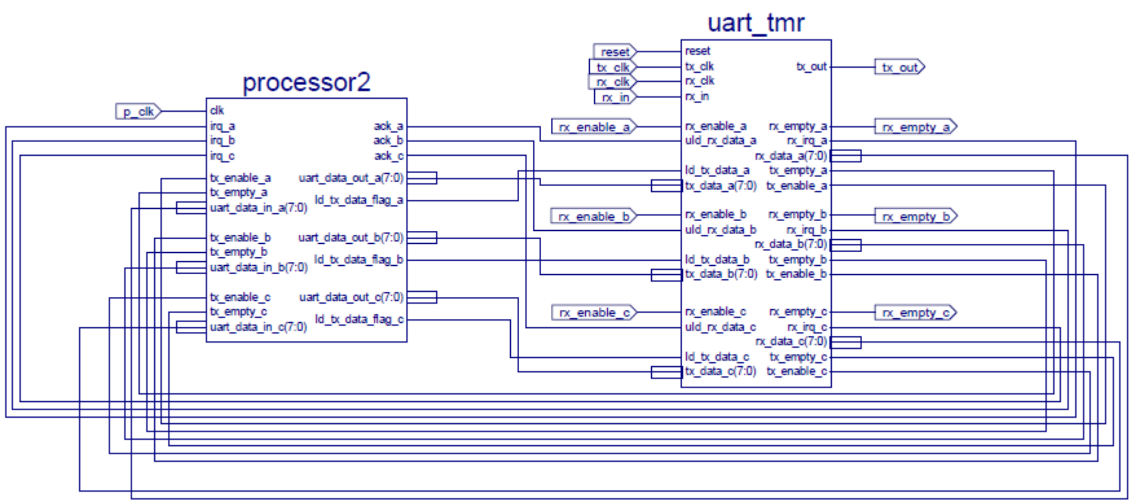


Figure 28. Schematic view of the UART's connection to the payload processor.

2. Receive Logic

The receive logic of the UART is initiated by passing a receive enable signal, indicating the UART should be prepared to receive data immediately. The UART sets the

receive busy signal high if it is not currently receiving data to indicate it is expecting incoming data. Next, the UART initiates a 4–bit sampling counter that is incremented on each cycle of the receive clock. The purpose of this counter is to determine when the UART should sample the input signal to obtain data. Sampling typically occurs near the middle of the input signal time interval to ensure distortion from signal transitions does not cause a bit error [31]. The UART on the payload processor is sampled on the eighth of 16 sample counter increments. Once the signal has been sampled, the resulting bit is appended to a receive register, and a receive counter is incremented. The receive counter samples ten values (bits 0 – 9), after which it sets the receive busy signal to low. The UART uses the first and last receive counter increments to indicate the start or termination of a transaction. Increments 1 – 8 are where the data bits are collected. Upon completion of the ten increments, the data is available to be transferred from the UART to the processor data memory.

3. Transmit Logic

The transmit logic of the UART is initiated by setting the load transmit data signal high. This indicates there is data to be transmitted by the UART that must first be copied into the transmit register. Once data transfer to the transmit register is complete, transmission commences when the transmit enable bit is set high. If the transmit enable bit is set low, the data waits inside the register. The parallel data is transmitted with a zero bit as the preamble, eight data bits, and is finished with a one bit on the tenth clock cycle.

4. Interface with the Payload Processor

The UART transmit and receive functionality must be capable of interfacing with the payload processor. When the UART receives data to pass to memory, it first transmits a set of three IRQ signals to **pc_select_module_trip** as shown in Figure 28. The **pc_select_module_trip** component sees these flags during the following clock cycle and automatically switches the new PC address to the IRQ start address since an IRQ has the highest priority. It then saves either the branch/jump address or the PC plus four address into a holding register depending on whether the **br_jump_sel** family of flags are set

high or low, respectively. This maintains the processor's place in instruction memory for when the ISR terminates. The PC selector then responds to the UART with an acknowledgement signal, causing the UART to unload its received data as input to the **data_mem_trip** module in the MEM stage. The data is not immediately written into memory after the unload signal, as the **pc_select_module_trip** first changes the PC address to the first instruction of the ISR. Four NOP instructions are used to clear the pipeline, followed by a store word instruction. The arguments of the store word instruction are zeroes as the interrupt request processing flag generated by **pc_select_module_trip** provides indication to the data memory on which input data to write to memory.

After the NOP instructions have cleared the pipeline, the interrupt request flag propagates to the MEM stage where it is received by data memory. Like instruction memory, data memory is divided into three equal segments of 45 kB each; however, data memory is further subdivided into two separate address spaces within each of the three 45 kB segments. One address space is for standard processor data manipulations. The other is dedicated to transactions involving the UART. Byte addresses 0 to 22,499 are used for standard read and write operations. Byte addresses 22,500 to 44,999 are reserved for the UART. The UART writes only a single byte as opposed to a word during an ISR because that is the most data that can be received during a single transaction with the interface. The data received by the UART is already present at the **data_mem_trip** module because the **pc_select_module_trip** acknowledged receipt of the original IRQ which triggered the UART to unload its data. The IRQ processing flag triggers the data memory module to take input from the UART data input interface vice the normal input address interface. The data memory module then references an internal register that stores the next appropriate byte address to which the UART data should be written. The UART input data is written to this location and the address register is incremented for the next receive transaction.

Transmit operations require significantly less overhead from the processor. The last index in the UART reserved address space (44,999) is dedicated for transmit operations. This address is the only location in the reserved address space to which the

processor writes data outside of an ISR. A load word instruction sends this data to the UART as well as writes its contents as the eight least significant bits of the register specified in the instruction. The **data_mem_trip** module also sets the transmit enable flag to high when the data is sent.

D. CHAPTER SUMMARY

The overall design and functionality of the payload processor was discussed in this chapter. The processor pipeline was considered first. Following conventional MIPS design, we saw that it consists of five stages. These five stages contain combinatorial logic modules that operate on smaller components of the 32–bit MIPS instructions. The outputs of these modules are passed onto pipeline registers where they are held until the following clock cycle and ITMR voting is performed. The IF stage translates a PC memory address into an instruction. The ID stage decodes the individual components of this instruction and sets the appropriate flags to control the pipeline modules. The EX stage contains the ALU and calculates the branch address. The MEM stage holds the data memory module and determines if a new branch or jump address should be sent to the PC selector module. Finally, the WB stage selects between the ALU result and data memory output to forward back to the register file.

The UART design and functionality was also considered in this chapter. It is capable of receiving or transmitting a single byte per transaction. The receive logic is interrupt–driven and uses an ISR to clear the pipeline and write the data to a reserved memory space. The transmit logic does not generate an IRQ but requires data to be written a memory address space that is specifically mapped for UART transmissions. A load word instruction then passes the memory contents to the UART for transmission.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TESTING AND ANALYSIS OF THE PAYLOAD PROCESSOR AND UART

The focus of this chapter is on testing the processor at three levels of its logic design. First, the pipeline register and voter functionality is verified by ensuring each of the pipeline registers successfully votes the majority signal. Next, each major category of instruction within the ISA is individually run through the processor to ensure it performs the correct operations, triggers the correct flags, and operates on the correct data. Finally, the UART is integrated into the system, and the receipt and transmission of data on the interface is verified. Each of these simulations was performed using Xilinx's ISim[3] software.

A. PIPELINE REGISTERS

Functionality of the pipeline registers were tested first to ensure the processor is capable of voting out simulated errors. Test cases were introduced to the registers in a Gray code sequence displayed in Table 7. This sequence exercises all eight combinations of the majority voters in the circuit while allowing only one transition per time interval amongst the three members of the set. Four of these combinations contain majority zeroes, and four contain majority ones. Output waveforms of the PC pipeline register are displayed in Figure 29. The output waveforms of all the pipeline registers can be found in Appendix B. The three rows above the clock signal in Figure 29 (**npc_a**, **npc_b**, and **npc_c**) represent the output of each member of the triplicated set of registers. The three rows below the clock signal (**pc_a**, **pc_b**, and **pc_c**) represent input from each of the members of the triplicated set of registers. It is important to note that for each combination of inputs, the register module outputs the majority signal on all three registers during the low to high transition of the clock signal. This confirms the correct operation of the pipeline registers and allows consideration of only one of the three parallel processors during the next stage of testing.

[3] ISim® is a registered trademark of Xilinx, Inc.

Table 7. Gray code sequence of inputs to the pipeline registers

Sequence	Bit A	Bit B	Bit C	Majority
1	0	0	0	0
2	1	0	0	0
3	1	1	0	1
4	0	1	0	0
5	0	1	1	1
6	1	1	1	1
7	1	0	1	1
8	0	0	1	0

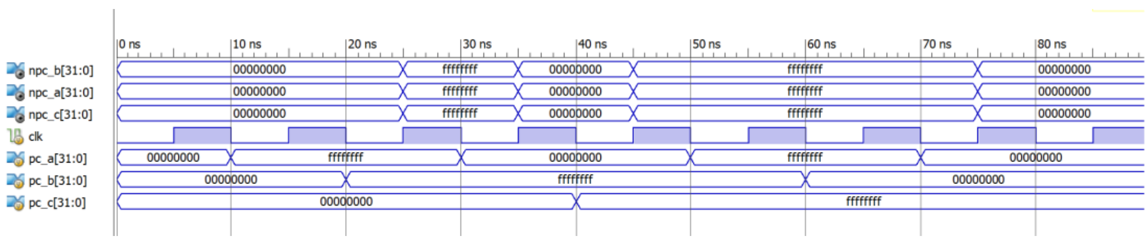


Figure 29. Waveform outputs of the triplicated PC register.

B. TEST PROGRAM EXECUTION

The purpose of this testing was to ensure all 24 instructions currently supported by the ISA are executed correctly. This was accomplished by developing an assembly code program that was placed in memory. By providing only the clocking signal and default UART inputs, the processor must run independently. The instruction sequence used to complete this testing is displayed in Table 8. Since instruction memory is byte–addressable, and each new instruction consists of four bytes, instruction address start on multiples of four. Several NOP instructions were required in succession to ensure data hazards were not encountered while register and memory locations were being read and written.

Table 8. Assembly program used for processor testing.

Memory Address	Instruction (Assembly Code)	Instruction (Machine Hexadecimal	Instruction Format
0	ADDI $0, $1, 0X000A	0x2001000A	I
4	ADDI $0, $2, 0xFFF0	0x2002FFF0	I
8 – 12	NOP (x2)	0x00000000	N/A
16	AND $1, $2, $3	0x00221824	R
20	OR $1, $2, $4	0x00222025	R
24	NOR $1, $2, $5	0x00222827	R
28	XOR $1, $2, $6	0x00223026	R
32	ADDI $0, $7, 0x0006	0x20070006	I
36	SLL $0, $7, $8	0x00074080	R
40	SRL $0, $8, $7	0x00083902	R
44	SLLV $0, $7, $8	0x00074004	R
48	SRLV $0, $8, $7	0x00083806	R
52	ADD $1, $2, $9	0x00224820	R
56	SUB $1, $2, $10	0x00225022	R
60	ANDI $2, $11, 0xFFFF	0x304BFFFF	I
64	ORI $2, $12, 0x0000	0x344C0000	I
68	SLT $9, $10, $13	0x012A682A	R
72	SLT $10, $9, $14	0x0149702A	R
76	SLTI $9, $15, 0x0004	0x292F0004	I
80	SLTI $10, $16, 0x0004	0x29500004	I
84	LW $0, $17, 0x0000	0x8C110000	I
88	SW $0, $11, 0x0000	0xAC0B0004	I
92	BEQ $0, $0, 0x000A	0x1000000A	I
96 – 132	NOP (x10)	0x00000000	N/A
136	BNE $1, $2, 0x000A	0x1422000A	I
140 – 176	NOP (x10)	0x00000000	N/A
180	BLEZ $17, 0x000A	0x1A20000A	I
184 – 220	NOP (x10)	0x00000000	N/A
224	BLZ $5, 0x000A	0x04A0000A	I
228 – 264	NOP (x10)	0x00000000	N/A
268	BGEZ $2, 0x000A	0x0441000A	I
272 – 308	NOP (x10)	0x00000000	N/A
312	BGZ $4, 0x000A	0x1C80000A	I
316 – 352	NOP (x10)	0x00000000	N/A
356	LW $0, $0, 0xAFC4	0x8C00AFC4	I
360	LW $0, $18, 0x57E4	0x8C1257E4	I
364 – 372	NOP (x3)	0x00000000	N/A
376	J 0x000000	0x08000000	J
380 – 388	NOP (x3)	0x00000000	N/A

Due to the number of signals present in the processor and duration of the simulation, the outputs of the ISim timing diagram are summarized in this section. A complete ISim timing diagram for the test program is included in Appendix B. Additionally, since the fault–tolerant capabilities of the pipeline registers have already been verified in Section A of this chapter, only one of the three parallel processors (processor A) is considered. For each instruction, it is critical the control flags are appropriately set during the ID stage. This ensures activation of the appropriate combinatorial logic modules by the remaining stages proceed as expected. I–format instructions must select the IMM operand over a second register operand, obtain the correct ALU result, and write–back to the register. R–format instructions must choose the correct write–back register, obtain the correct ALU result, and write–back the result to the register during the appropriate clock cycle. Load word and store word instructions must reference the appropriate register file and memory addresses and show that these locations were written correctly. Finally, the branch and jump instructions must exhibit successful change of the next PC address at the PC selector module in the MEM stage. Additionally, branch instructions must exhibit the correct comparison between the branch flags from the ID stage and the comparison bits from the EX stage.

1. I–Format Instructions

The ANDI and ORI instructions (memory addresses 60 and 64, respectively) and their propagation through the pipeline are evaluated during this section. First, the ID stage outputs of these instructions are displayed in Figure 30. The instruction being operated on at any point by the ID stage is denoted in the inst_a row of the timing diagram and can be cross–referenced with the hexadecimal format in Table 8. The ANDI instruction is present in the ID stage from 32 – 34 ns and the ORI instruction from 34 – 36 ns as shown in Figure 30. The **faluop_a** is initially set to two and transitions to three at 34 ns, which correlates with the ALU operations listed in Table 5. The IMM field of each instruction is sign–extended and forwarded to a multiplexer in the EX stage. Only the **falusrc_a** and **fregwrite_a** flags are set for I–format instructions. The **falusrc_a** flag triggers the EX stage multiplexer to select the sign–extended IMM value over the second

register operand. The **fregwrite_a** flag serves as a write enable signal when the result is written back to the register file during the WB stage.

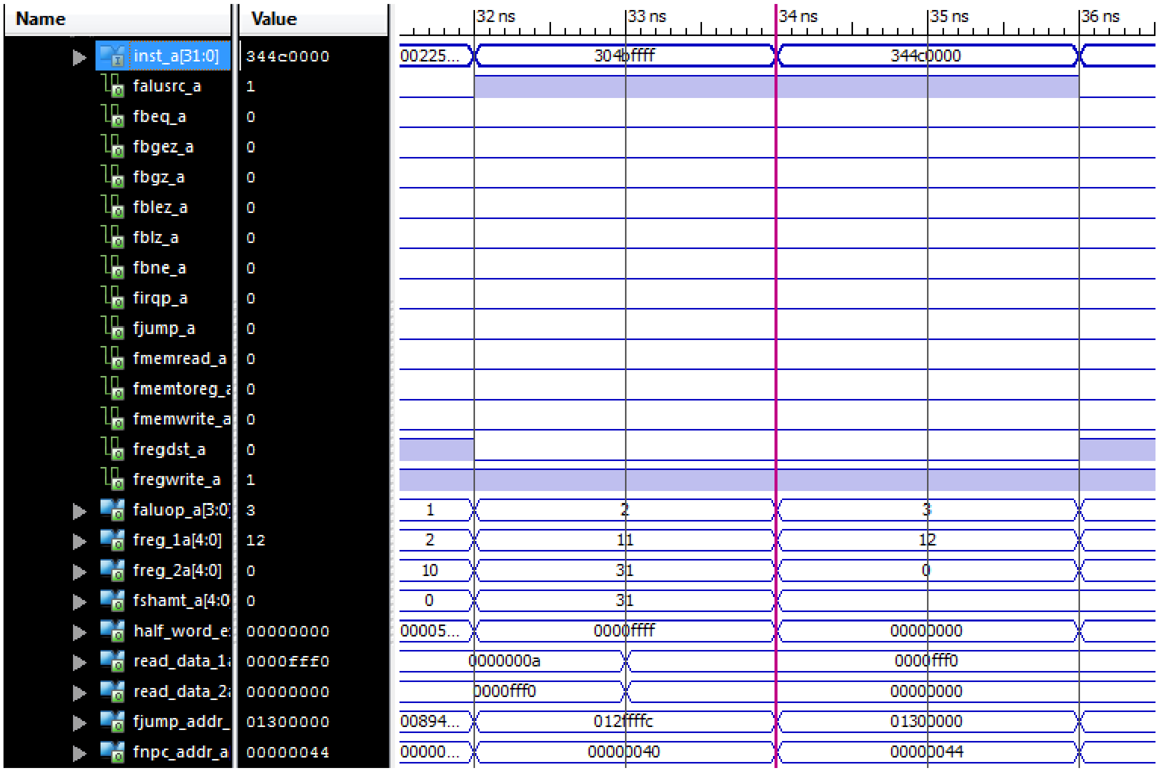


Figure 30. ID stage outputs for the I–format instructions ANDI, and ORI.

The EX phase performs three critical functions for I–format instructions. First, the ALU input multiplexer must select the sign–extended IMM field as the second ALU operand. Second, the ALU must produce the correct mathematical result. Finally, the write–back address multiplexer must determine the correct register to which the result is forwarded. Each of these tasks is verified in Figure 31. The second operand to the ALU is 0x0000FFFF for the ANDI instruction and 0x00000000 for the ORI instruction. Both of these operations produce the result 0x0000FFF0 displayed in the **alu_result_a** row. The **wb_reg_addr_a** row indicates the result of the operation is written to registers $11 and $12.

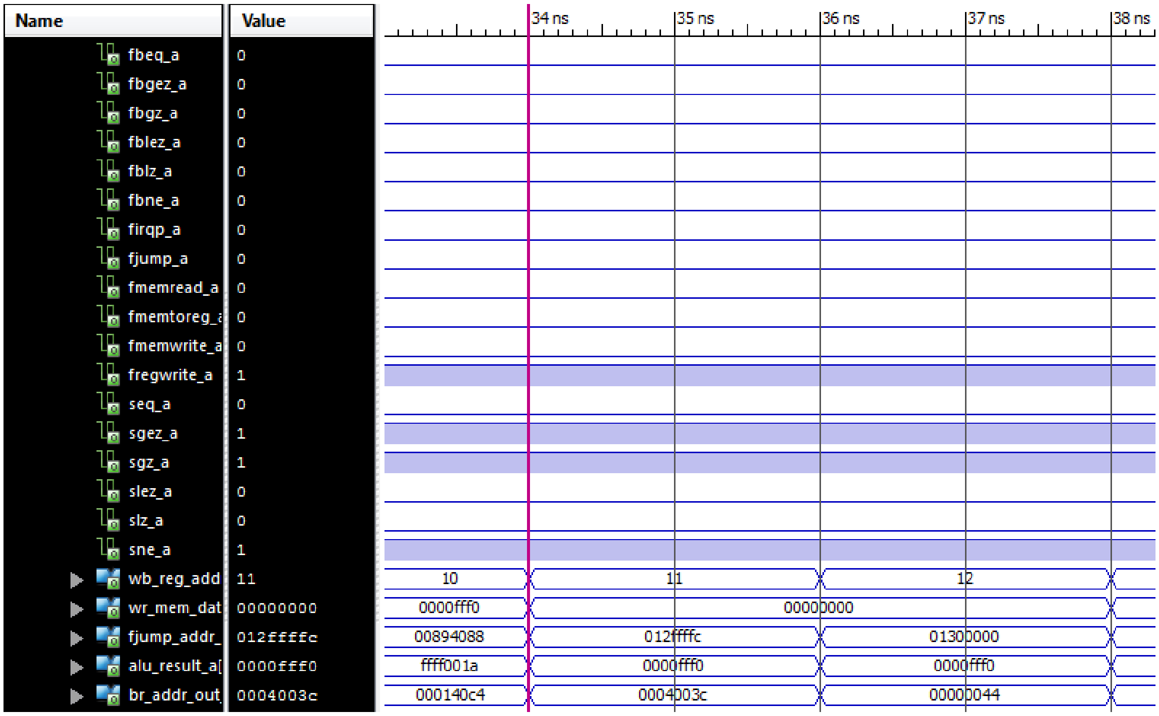


Figure 31. EX stage outputs for the I–format instructions ANDI and ORI.

In the WB stage, the WB multiplexer selects the correct data source to write to the register file. This data is written to the register file during the same clock cycle. This function is displayed in Figure 32. The WB multiplexer data changes at 38 ns of simulation time to the ALU result of 0x0000FFF0, previously established as the correct result of the ANDI and ORI instructions. The data_out11 row displays the output of the register $11 and transitions from 0x00000000 to 0x0000FFF0 at 38 ns. Likewise, register $12 makes the same transition at 40 ns, one complete clock cycle after the write–back of the ANDI instruction is complete. The undefined red signal shown in the **input2_a** row of Figure 32 originates from the data memory module. This signal obtains a defined value when a data memory read operation via a load word instruction is processed.

Figure 32. WB stage outputs for the I–format instructions ANDI and ORI.

2. R–Format Instructions

Propagation of the AND, OR, NOR, and XOR instructions (memory addresses 16, 20, 24, and 28, respectively) are considered in this section. The ID stage outputs for this subset of instructions are shown in Figure 33. From 10 – 18 ns the **faluop_a** signal cycles sequentially through values two, three, five, and four. Referencing Table 5, we see that these ALU operations correspond to the AND, OR, NOR, and XOR sequence desired. The two flags set high are **fregdst_a** and **fregwrite_a**. The **fregdst_a** flag indicates the destination register is contained in the **rd** register field, which is the correct setting for R–format instructions. The **fregwrite_a** flag has the same functionality displayed during the WB stage of the I–format instructions discussed in the Subsection 2 of this section.

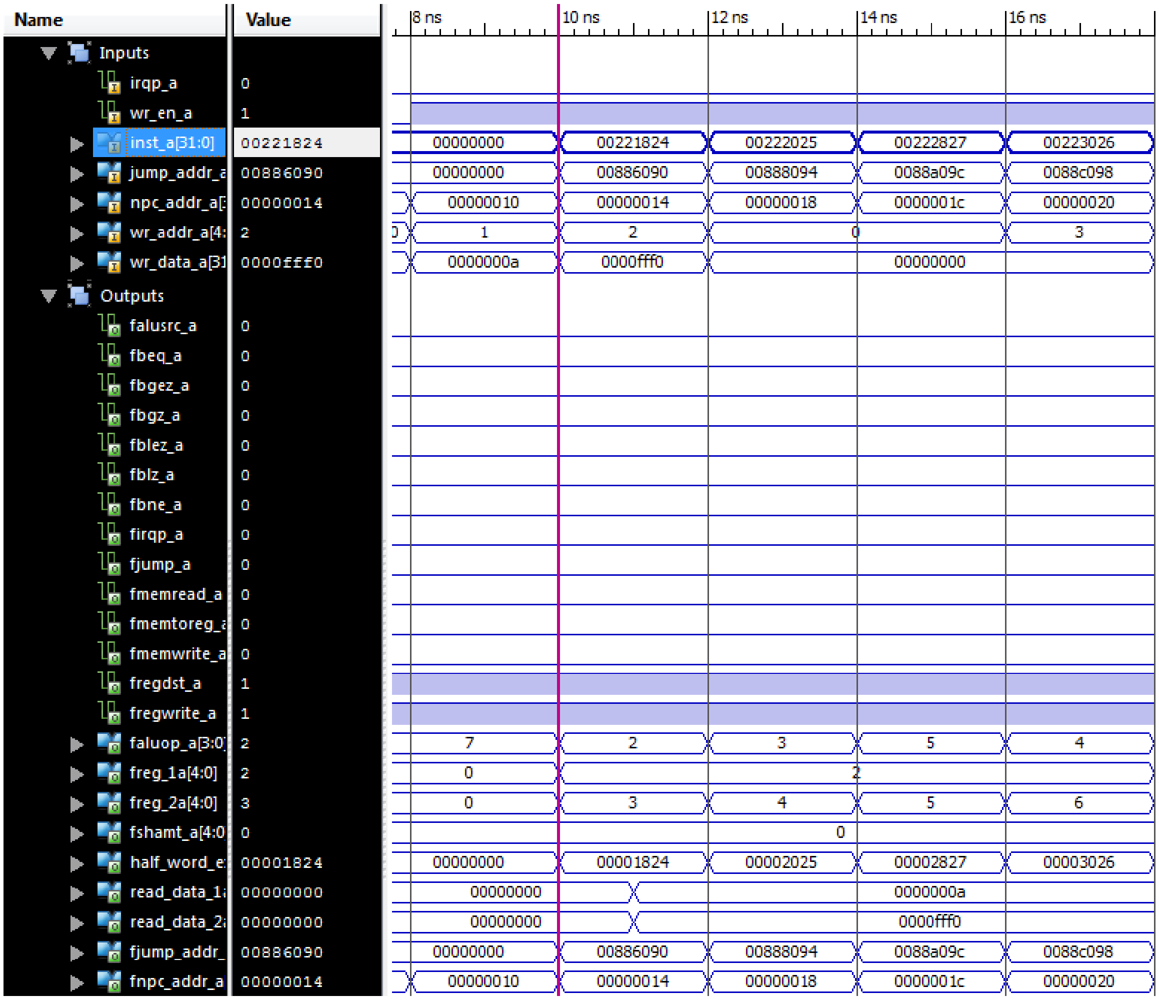


Figure 33. ID stage inputs and outputs for the R–format instructions AND, OR, XOR, and NOR.

In the EX stage, each of these instructions takes the data contained in registers $1 (0x0000000A) and $2 (0x0000FFF0) as operands. The result of each logical operation is shown by the **alu_result_a** row in Figure 34. Only the **regwrite_a** control flag is forwarded from this stage (displayed as **fregwrite_a** in Figure 34). The **regdst_a** flag is terminated in the EX stage when it is used as a selector bit to the write–back register address multiplexer. It is also important to note the comparison bits **sne_a**, **sgz_a**, and **sgez_a** are set high following the output of all four R–format instructions because the register values for each of them are the same. Since no branch flags are set during the ID stage, the processor does not trigger a branch/jump selector flag during the MEM stage.

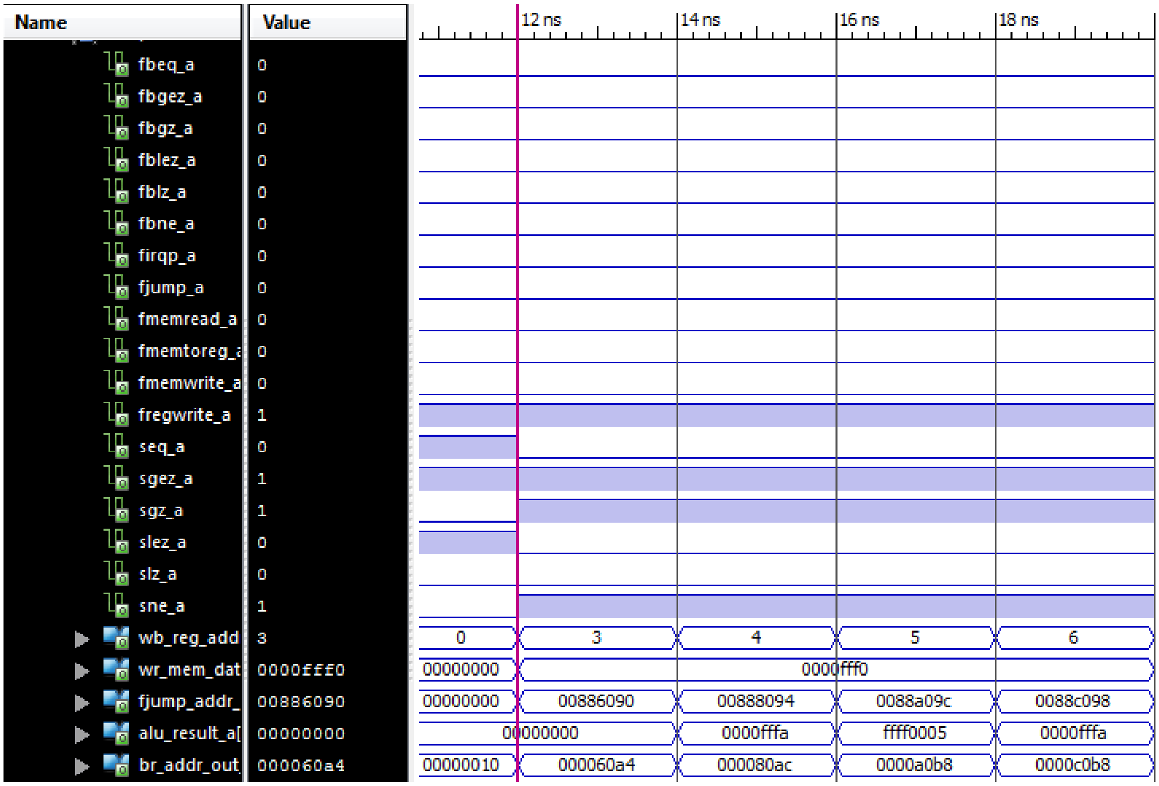


Figure 34. EX stage outputs for the R–format instructions AND, OR, XOR, and NOR.

Finally, the WB stage of the R–format instruction is evaluated. This stage of execution is shown in Figure 35. Identical to an I–format instruction, the WB multiplexer first chooses between the ALU result and data memory output to be written into the register file. The **sel_a** input, controlled by the **memtoreg_a** flag is set low, which ensures the write–back data originates from the ALU. During the positive clock edges occurring from 16 – 24 ns, the WB multiplexer output, **data_out_a**, matches the data received as input to the register file. The **input2_a** signal is still undefined at this point in the simulation because the first load word instruction has not yet entered the MEM stage.

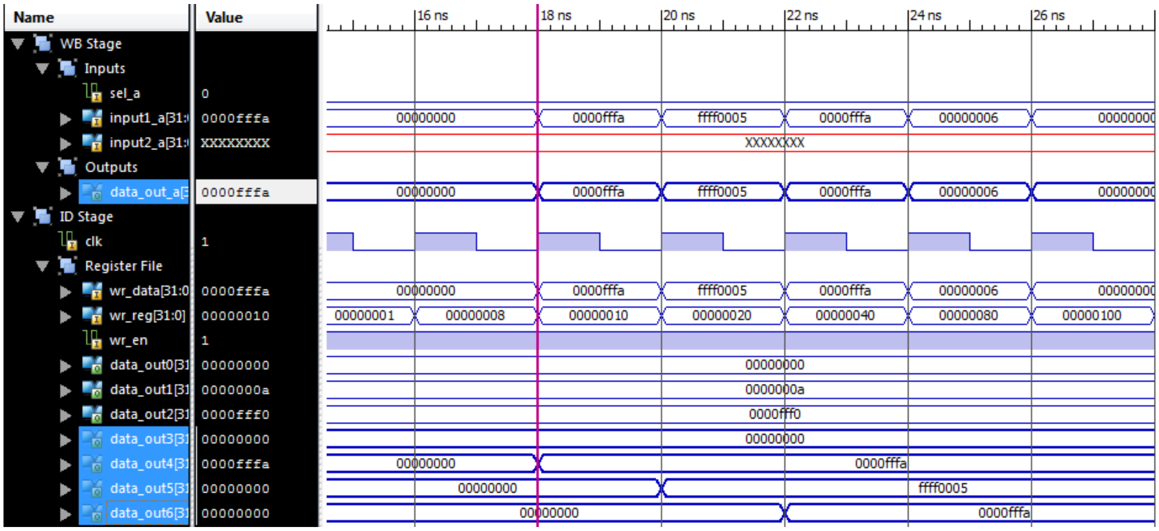


Figure 35. WB stage outputs for the R–format instructions AND, OR, XOR, and NOR.

3. Load Word and Store Word Instructions

The load word and store word instructions are currently the only instructions in the ISA that interact with data memory. They essentially have the same structure as I–format instructions, but the control flags they trigger force different outputs, particularly during the MEM and WB stages. There is one of each load word and store word instruction in the testing program located at addresses 84 and 88, respectively. The ID stage outputs for each of these instructions are displayed in Figure 36.

The load word instruction is present in the ID stage from 43 – 45 ns of simulation time. Decoding this instruction produces the registers $0 in the **reg_1a** row and $17 in the **reg_2a** row, a zero in the **aluop_a** row indicating addition, and an IMM field with the offset value 0x00000000 in the **half_word_ext_a** row. Additionally, the **fregwrite_a**, **falusrc_a**, **fmemread_a**, and **fmemtoreg_a** control flag rows are set high. The **fmemread_a** flag serves as a read enable signal to the data memory module in the MEM stage. The **memtoreg_a** flag is forwarded to the WB multiplexer where it selects the output of the data memory module over the result of the ALU to write to the register file.

The store word instruction is present in the ID stage from 45–47 ns of simulation time. This instruction decodes **reg_1a** as $11 and **reg_2a** as $0, zero as the **aluop_a**

output, and 0x00000004 for the offset. The store word instruction sets the **falusrc_a** and **fmemwrite_a** flags from the control module. The **fmemwrite_a** flag is forwarded to the MEM stage where it functions as a write enable bit for the data memory module.

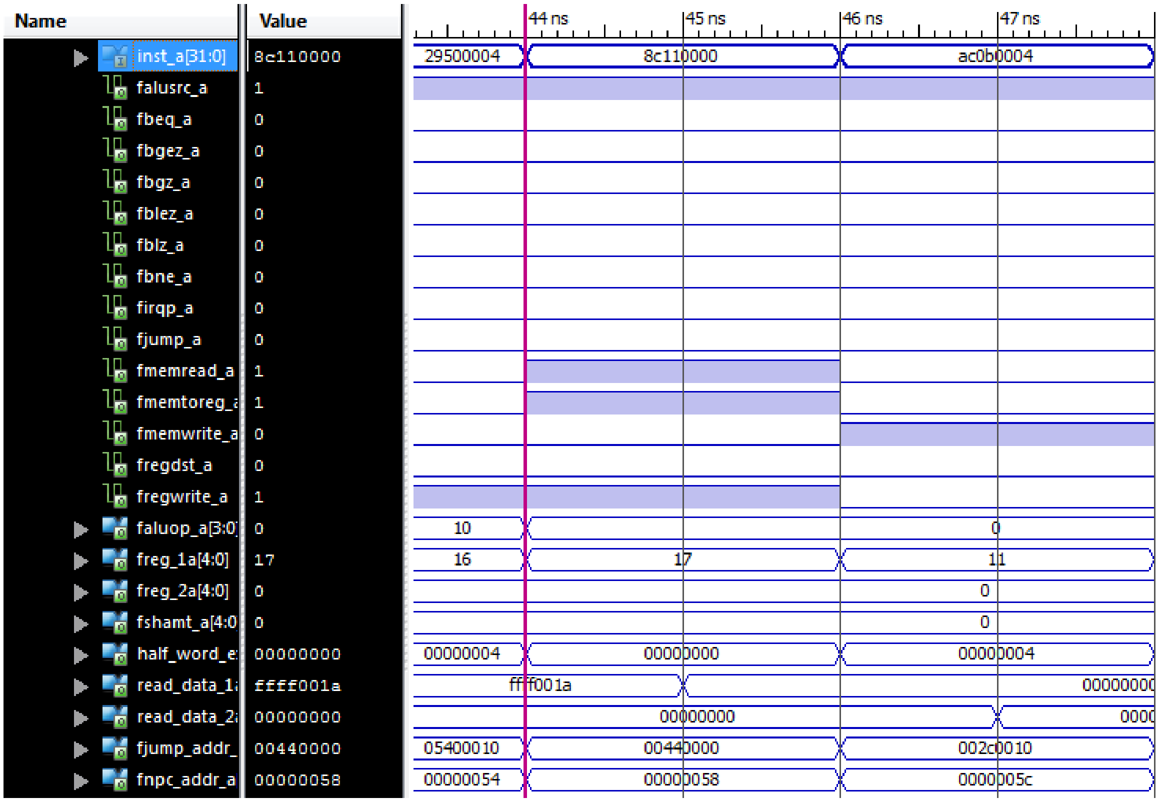


Figure 36. ID stage outputs for the load word and store word instructions.

In the EX stage, the sign–extended IMM operand is added to the first register operand to obtain the data memory address which is read or written. This action is shown in Figure 37 where **opa_a** and **opb_a** are the base register and sign–extended offset, respectively. These results are added to obtain the data memory address 0x00000000 and 0x00000004 displayed in the **alu_result_a** row. This simulation uses data memory address 0x00000000 for the load word instruction and address 0x00000004 for the store word instruction. Prior to the start of the simulation, each byte in data memory was initialized to zeroes with the exception of the first four byte addresses which contain ones; thus, a read operation performed on the first byte of data memory writes a value of 0xFFFFFFFF back to the register file.

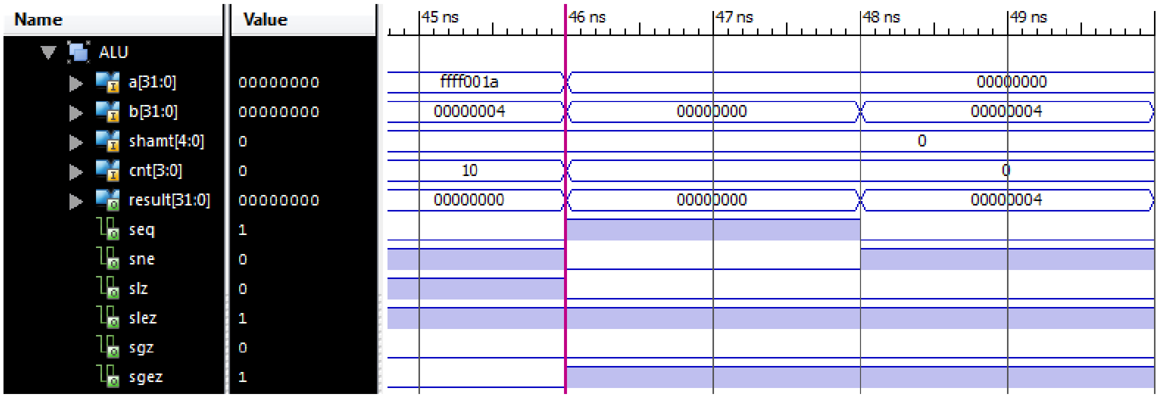


Figure 37. EX stage outputs for the load word and store word instructions.

The MEM stage is where the load word and store word instructions perform the unique operation of interacting with data memory. The MEM stage timing diagram for the load word and store word instructions is displayed in Figure 38. First, the load word instruction enters the MEM stage at 48 ns in simulation time. The **wr_en** signal is set high because it is linked to the **memread_a** control flag set in the ID stage. The **data_out** row of the data memory module transitions from an undefined output to the aforementioned value 0xFFFFFFFF expected at the memory location 0x00000000.

The store word instruction enters the MEM stage at 50 ns in simulation time. Just as the **re_en** signal was linked to the **memread_a** control flag, the **wr_en** signal receives the forwarded **memwrite_a** control flag as input. This causes the inversion of each of the **re_en** and **wr_en** signals on the load word and store word instructions. Data memory is too large to display in the ISim software; however, since the memory location was only written once during the simulation, a successful write can be confirmed by checking the value column of the objects panel in ISim. This panel displays the first 64 byte addresses in data memory and their value at the end of simulation time. The status of this panel at the end of the simulation is displayed in Figure 39 and shows byte addresses four through seven written with the data 0x0000FFF0.

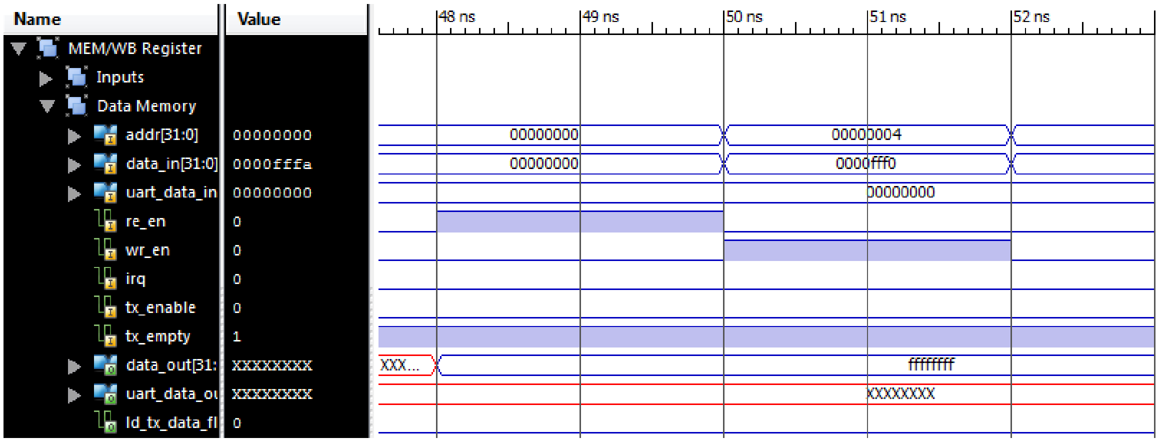


Figure 38. MEM stage outputs for the load word and store word instructions.

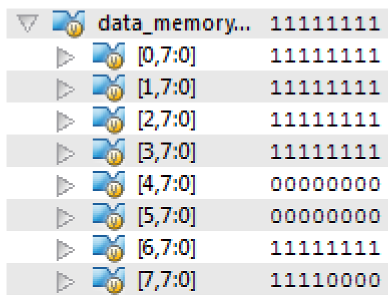


Figure 39. ISim object panel display of the first eight bytes of data memory after simulation.

The WB stage of the processor performs the same basic operation for the load word instruction as for the I–format and R–format instructions. For the load word instruction, data sent to the register file comes from the data memory module output vice the ALU. The WB stage timing diagram for the load word instruction in the processor test program is displayed in Figure 40. When the simulation reaches 50 ns, the **sel_a** bit in the WB multiplexer is set high, and the **data_out** row carries the data read from memory. During this same clock cycle, near the bottom of Figure 40, a transition occurs in **data_out17** of the register file indicating register $17 was written with the data from memory location 0x00000000.

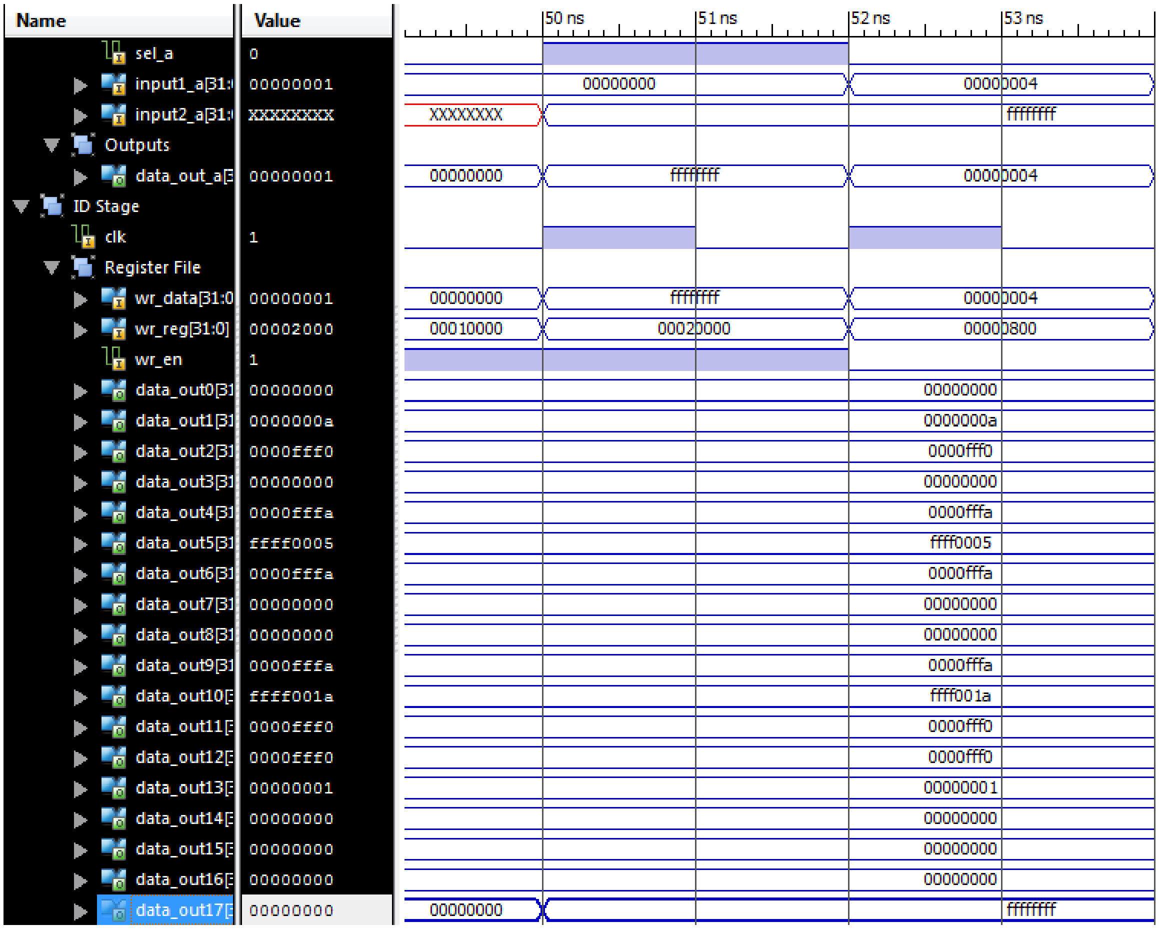


Figure 40. WB stage outputs for the load word and store word instructions.

4. Branch Instructions

Branch instructions, like load word and store word, essentially follow the same structure as I–format instructions. During the EX stage, branch instructions compare the **rs** and **rt** registers, but the ALU result is insignificant. The comparison bits from the ALU and branch flags from the ID stage are then compared in the MEM stage to determine if the requirements for a branch instruction have been met. If the branch criteria are met, the address and a branch flag are forwarded to the PC selector module. The PC selector module is then responsible for changing the PC to the branch address if the flag is set high. The branch–on–equal (BEQ) instruction located at address 92 of instruction memory is evaluated to show the functionality of branch instructions.

In the ID stage, branch instructions trigger at least one of the six possible branch control flags. The BEQ instruction triggers the **fbeq** flag at 160 ns into the simulation shown in Figure 41. No other control flags are set by branch instructions during the ID stage.

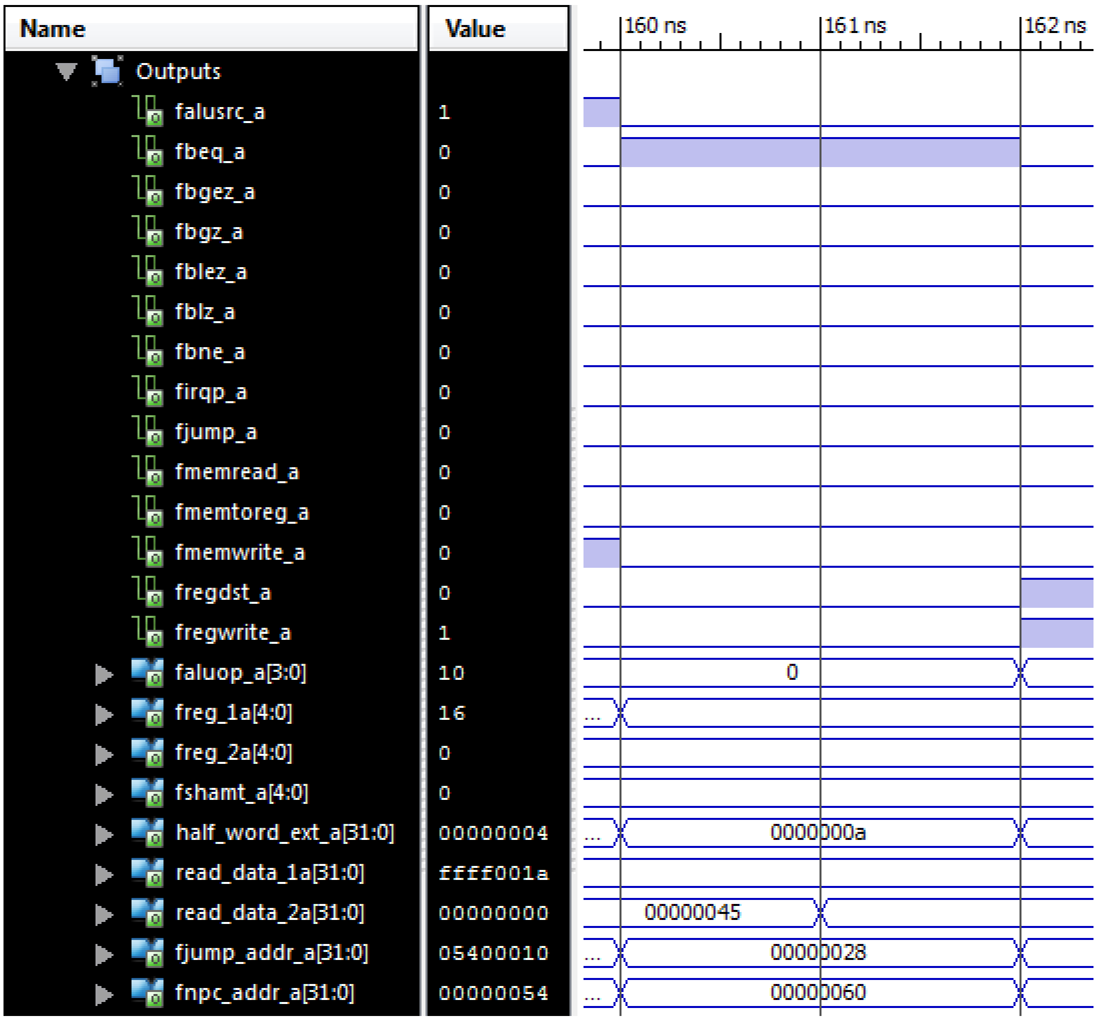


Figure 41. ID stage outputs for the BEQ instruction.

The ALU result calculated in the EX stage is not used in data memory or the WB stage. The ALU result is by–passed by setting the control flags **fmemread_a**, **fmemwrite_a**, and **fregwrite_a** low from 162 – 164 ns shown in Figure 42. In the BEQ instruction, both ALU operands are compared to determine the branch conditions have been met. These outputs are depicted in Figure 42 as **seq_a**, **sne_a**, **sgez_a**, **sgz_a**, **slez_a**,

and **slz_a**. Since both register operands have a value of zero, the **seq_a**, **sgez_a**, and **slez_a** bits are set high. These flags and comparison bits are further analyzed by the branch multiplexer and address selector modules during the MEM stage.

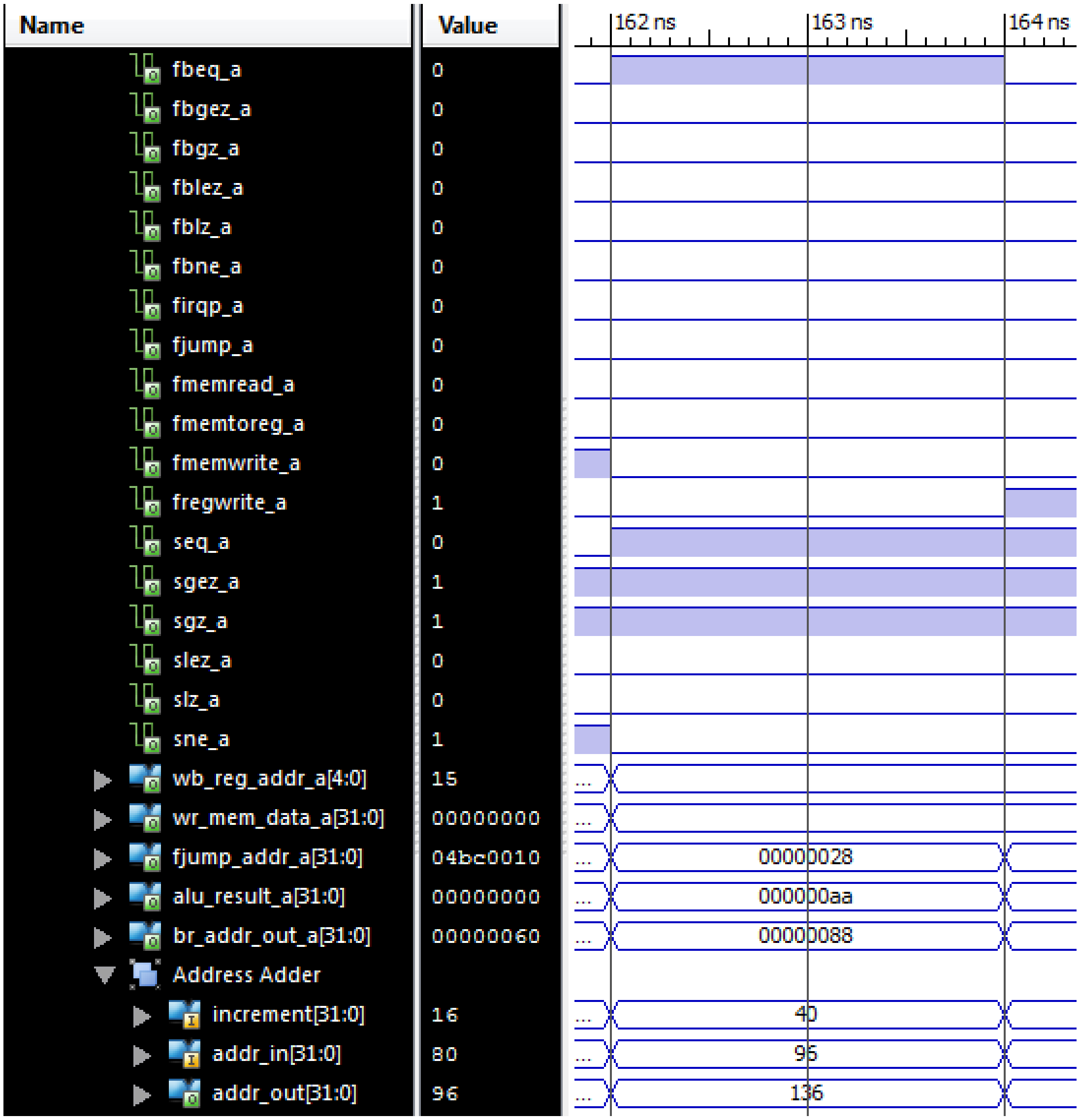


Figure 42. EX stage signals while executing the BEQ instruction.

The EX stage also calculates the branch address that is forwarded to the PC selector module. The memory offset included in the IMM field essentially represents the number of instructions that are skipped if the branch criteria are met. This value is first converted to a byte–addressable memory offset performed by logical left–shifting the

sign–extended IMM field by two bits, effectively multiplying the offset by four. This function is shown by the **addr_in**, **increment**, and **addr_out** rows in Figure 42. The offset from the IMM field, 0x0000000A, is converted from 10 words to 40 bytes in the increment row. This result is added to the **addr_in** row to obtain a branch address location of 136 which is forwarded to the MEM stage via the **addr_out** output.

The MEM stage is particularly important for branch instructions because of the operations performed by the address selector and branch multiplexer modules. Inputs and outputs of these modules are displayed in Figure 43. At 164 ns of simulation time the branch control flags and comparison bits from the EX stage enter the MEM stage. The branch multiplexer performs comparisons between each branch control flag and its corresponding comparison result from the ALU. For the BEQ instruction under consideration, the **beq** control flag and **seq** comparison bit are both set high during this clock cycle. This results in the branch output signal being set high and forwarded to the address selector module.

The address selector module takes the branch address obtained in the EX stage and branch output of the branch multiplexer as inputs. The branch input is represented in the **br_sel** row of Figure 43. A comparison is then made between the **br_sel** and **jump_sel** inputs that determines the appropriate address to forward to the PC selector module. The output **br_jump_sel_out** is essentially a XOR of the two bits since it is not possible to have a branch and jump flag in the same clock cycle. For the BEQ instruction under consideration, the **br_jump_sel_out** bit is set high, which causes address 136 to be placed on the **addr_out** connection to the PC selector module.

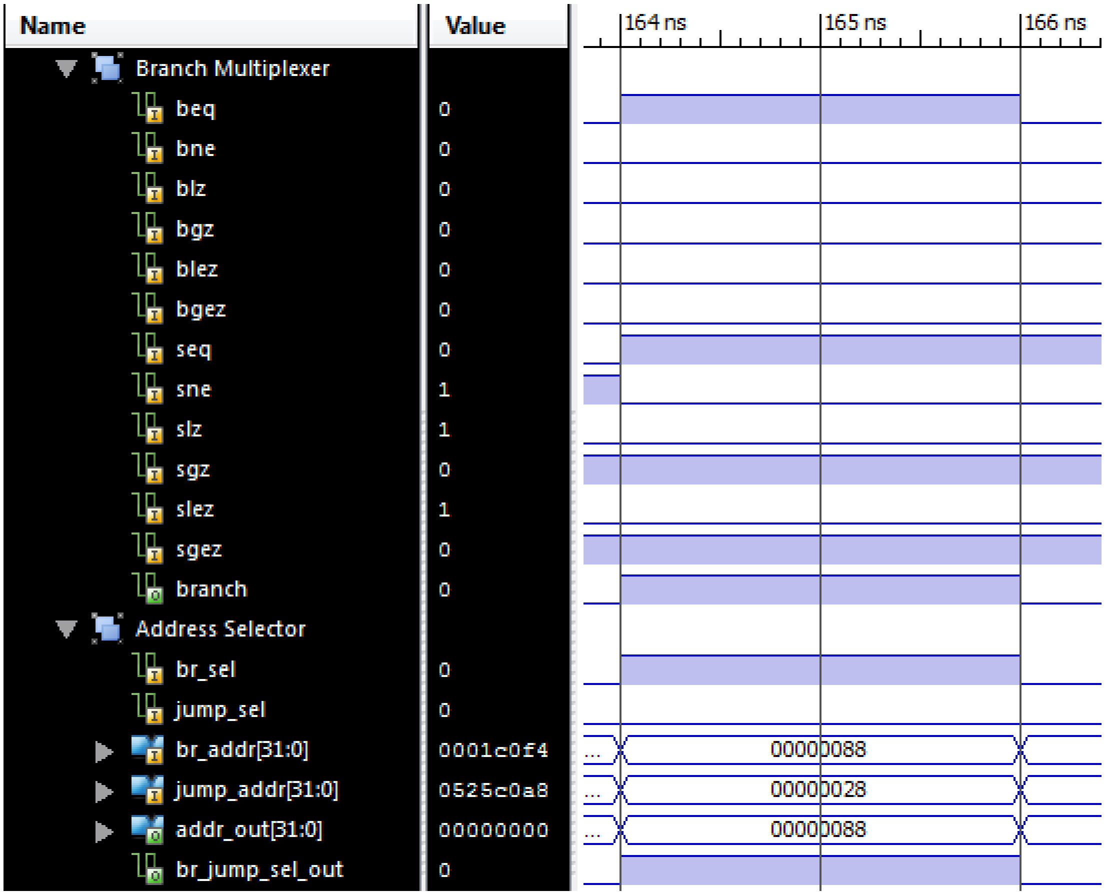


Figure 43. MEM stage inputs and outputs of the branch instruction.

Finally, the PC selector module detects the **br_jump_sel_out** signal and changes the PC address. The branch PC address is received at the **br_jump_addr** input shown in Figure 44. The **npc** address in the PC Selector grouping changes from 104 to 136 at 164 ns, indicating the branch address has been selected over the PC plus four address location. It is important to note the **br_jump_addr** and **br_jump_sel** flag inputs arrive at the PC selector module on the clock cycle spanning 164 – 166 ns. This indicates the result of the branch instruction is obtained immediately following any combinatorial logic delay experienced through the branch multiplexer and address selector modules. It is also shown in Figure 44 that two complete clock cycles must execute before the branch address is available in the PC register; thus, two NOP instructions must follow a branch to ensure pipeline hazards do not occur.

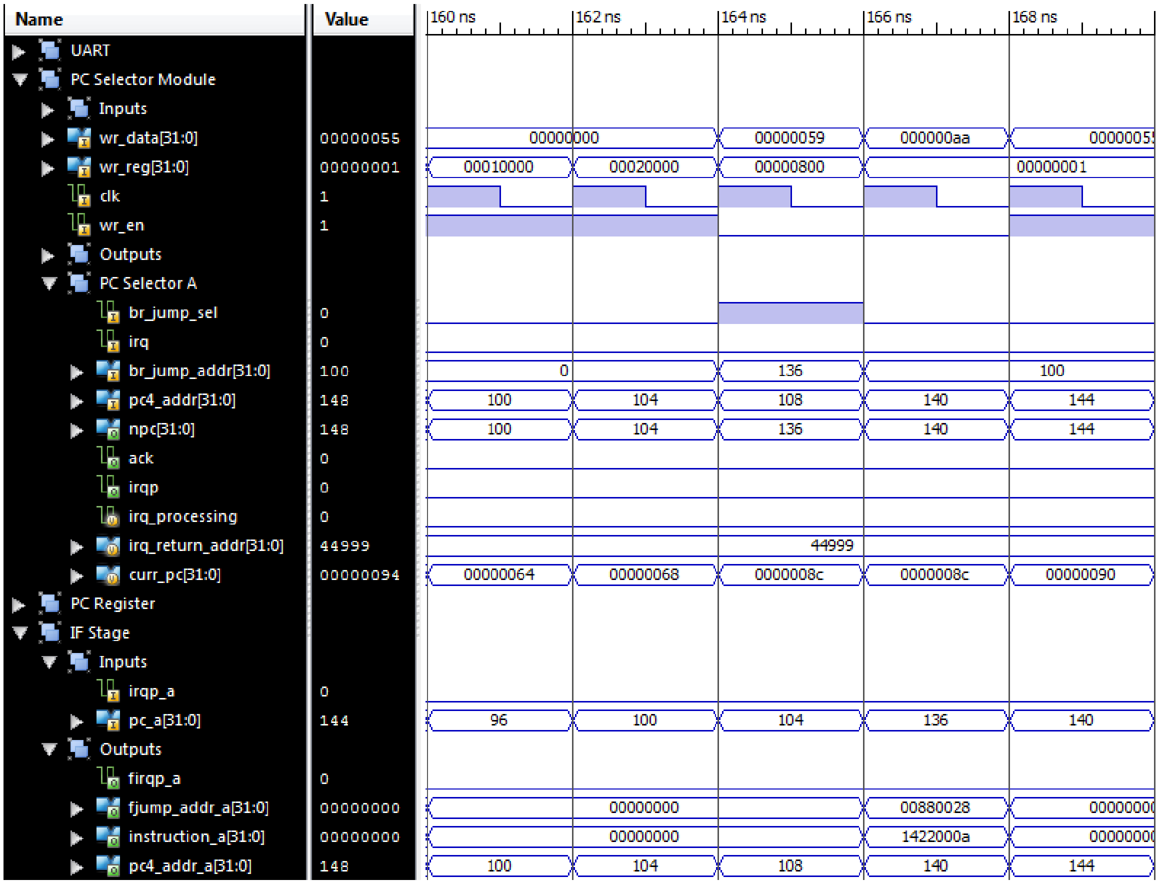


Figure 44. PC selector module and PC register transitions upon receipt of a branch address.

5. Jump Instructions

The jump instruction has a unique format but executes in a very similar manner to the branch instruction. The only jump instruction featured in the test program occurs at address 376, which is performed to restart execution of the test program again at address 0 in instruction memory. The majority of work associated with jump functions occurs in the IF stage. A jump address is formed by concatenating the four most significant bits of the PC plus four address, the 26 least significant bits of the jump instruction, and two zero bits. This operation is verified in Figure 45, which displays the **26–bit inst_in** and **4–bit npc** signals as inputs. The complete 32–bit output address is displayed in the **inst_out** row of Figure 45.

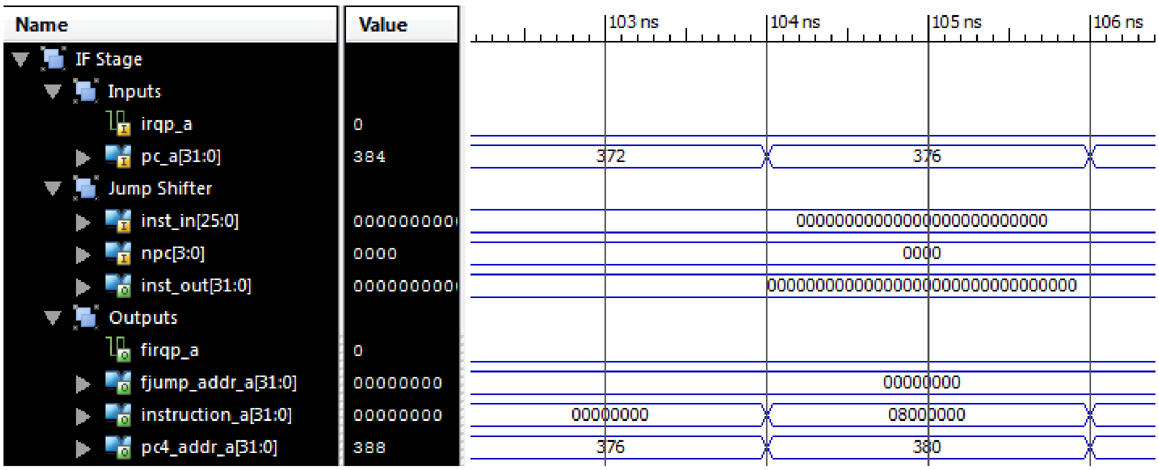


Figure 45. IF stage outputs of the jump instruction.

In the ID stage, the jump control flag is the only flag bit set for the jump instruction. The jump address and control flag are then forwarded to the address selector module in the MEM stage. Similar to the branch instruction, the address selector uses the jump control flag and jump address to forward the correct output back to the PC selector module. This output is verified in Figure 46. The **jump_addr** value is zero, and **jump_sel** flag bit is set high from 110 – 112 ns of simulation time. This produces an **addr_out** output of zero and sets the **br_jump_sel_out** flag set high. During the same clock cycle, the zero address is received at the PC selector module indicated by the **br_jump_sel_a** and **br_jump_addr_a** signals in Figure 46.

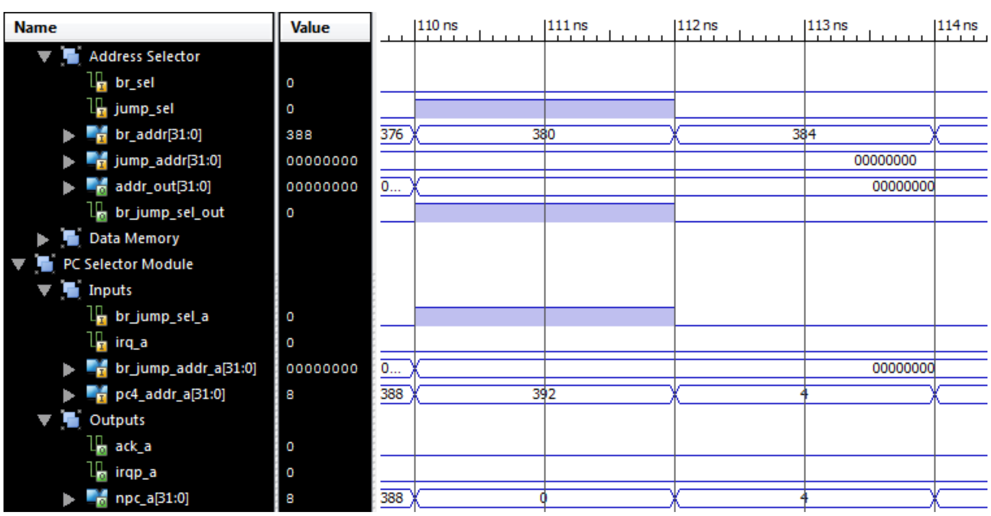


Figure 46. MEM stage and PC selector module outputs of the jump instruction.

C. UART TESTING

The UART is the last critical component of the payload processor requiring functional testing. The UART passes data received on its serial transmission connection to the pipeline via the PC selector module as discussed in Chapter IV; therefore, a majority of the testing for the receive function focuses on the PC selector module. The test program verifies successful placement of received UART data in memory using a load word instruction since the ISim software does not display more than the first 63 data memory addresses in its wave configuration panel. UART transmit verification was simpler to perform since the load word instruction referencing mapped memory initiates the transmit process. The load word instructions associated with transmitting and receiving UART data are located at addresses 356 and 360 of the test program, respectively.

1. UART Receive Testing

The UART is required to send an IRQ to the processor pipeline upon successful receipt of a single byte of data. Once acknowledgment is received from the PC selector module of the processor, the data is forwarded to memory where it remains until the interrupt processing flag arrives at the MEM stage. In order to verify this task, simulated data was sent on the serial connection line of the UART via a test bench file. Once the UART and processor perform their aforementioned tasks, the test program eventually executes the load word instruction at address 360, which places the received data into a register that can be viewed from the ISim software package.

The receipt of data from the serial interface begins when the **rx_enable** bit is set high. This action is performed at 2 ns into the simulation depicted by Figure 47, when the reset bit transitions from high to low. Signal sampling on the serial connection begins immediately following the low to high transition of the **rx_enable_a** signal shown in Figure 47. The receive and processor clocks were run at the same rate during this simulation. Typically, the receive clock operates at a much slower rate than the processor clock, but the two were matched to better view the simulation results. The receive clock required 16 clock cycles (32, 1 ns clock transitions) to obtain one bit. Ten total bits were

received, including the preamble zero and terminating one which remains on the **rx_in** line following receipt of the bits.

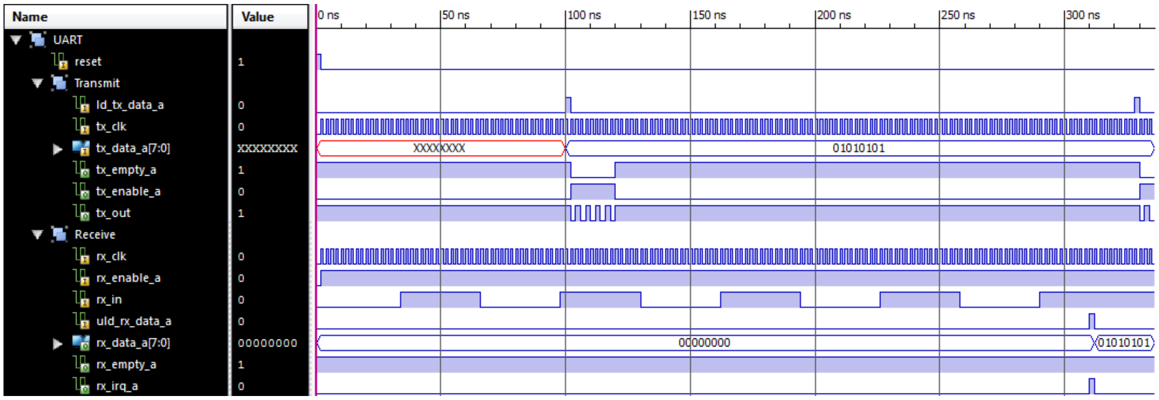


Figure 47. Inputs and outputs of the UART receive system during testing.

The signal is completely received by the UART following its final sample at 312 ns into the simulation as shown in Figure 48. Immediately following the tenth sample, the **rx_irq_a** interrupt flag is raised and sent to the PC selector module. The PC selector module recognizes the IRQ as the highest priority operation and immediately sends an acknowledgement via the **ack_a** signal back to the UART. The **ack_a** signal then triggers the **uld_rx_data_a** bit in the UART causing it to forward the received data to the data memory module. The PC selector module then places the processor in an IRQ processing state by setting the **irq_processing** flag bit high. The **npc_a** value is changed to the first address of the ISR in instruction memory, and the **curr_pc** register holds the second ISR address. This configuration of the **npc_a** and **curr_pc** values causes the processor to continue its ISR during the next clock cycle.

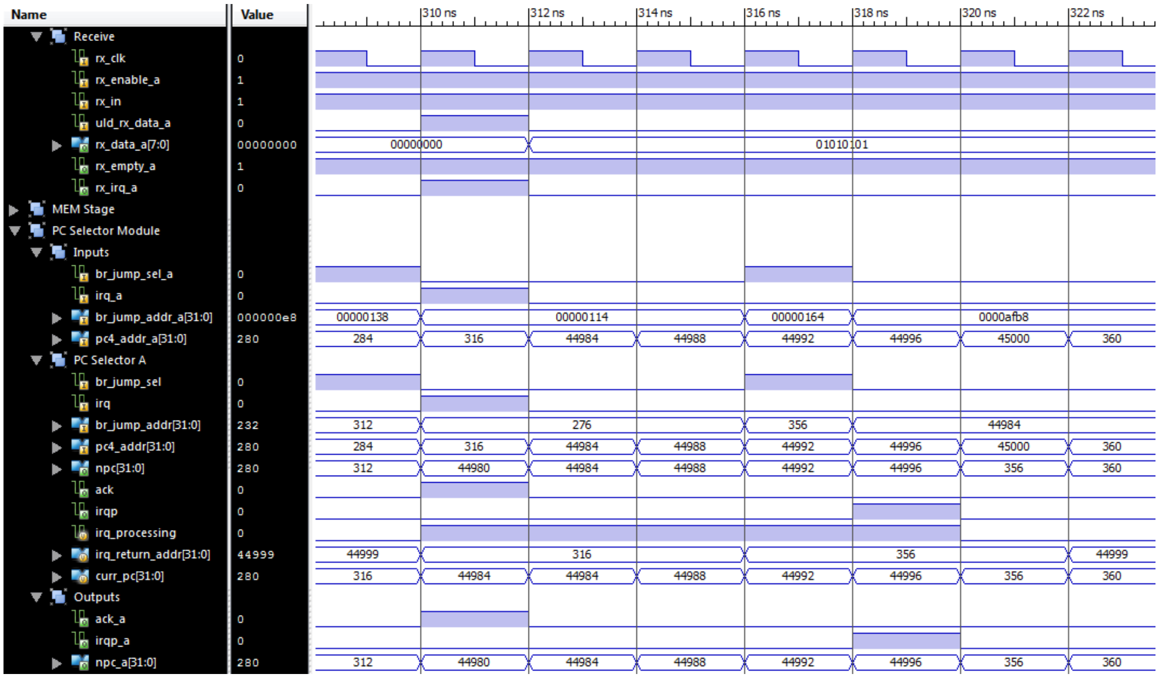


Figure 48. IRQ processing performed by the UART and PC selector module.

During the clock cycles from 312 – 316 ns, the PC selector module updates the **pc4_addr_a**, **curr_pc**, and **npc_a** values using the same method as a standard PC plus four increment. From 316 – 318 ns, while the ISR is using NOP instructions to clear the pipeline, the branch instruction at address 356 generates a new branch address and raises the **br_jump_sel_a** flag. Since a programmer does not know when the processor is performing an ISR, the PC selector module must update the value of the irq_return_addr register to the **br_jump_addr_a** value if the **br_jump_sel_a** flag is raised while the ISR is executing. When the ISR is performing its final instruction at address 44996, the PC selector module sets the **curr_pc** and **npc_a** values to address 356, which was the last address stored during the ISR. The irqp_a flag is also raised to accompany the store word instruction and propagates through the processor to the MEM phase.

When the **irqp_a** signal reaches the data memory module, it provides an indication to read input data from the UART interface vice the standard input interface. It is shown in Figure 49 at 328 ns the **irqp_a** flag bit enters the MEM stage where the UART data has been present since the PC selector module acknowledged the initial IRQ. During this clock cycle the data is written to address 22500, which is the first memory

address mapped to the UART. Upon completion of the memory write, the **uart_wr_addr** register is incremented to 22501 on the following clock cycle. If another UART write occurs during the simulation, it is written to this address. A load word instruction that reads the data in address 22500 immediately follows the UART write operation in data memory. The **data_out** signal indicates the read operation from this memory produces the data written by the UART on the previous clock cycle.

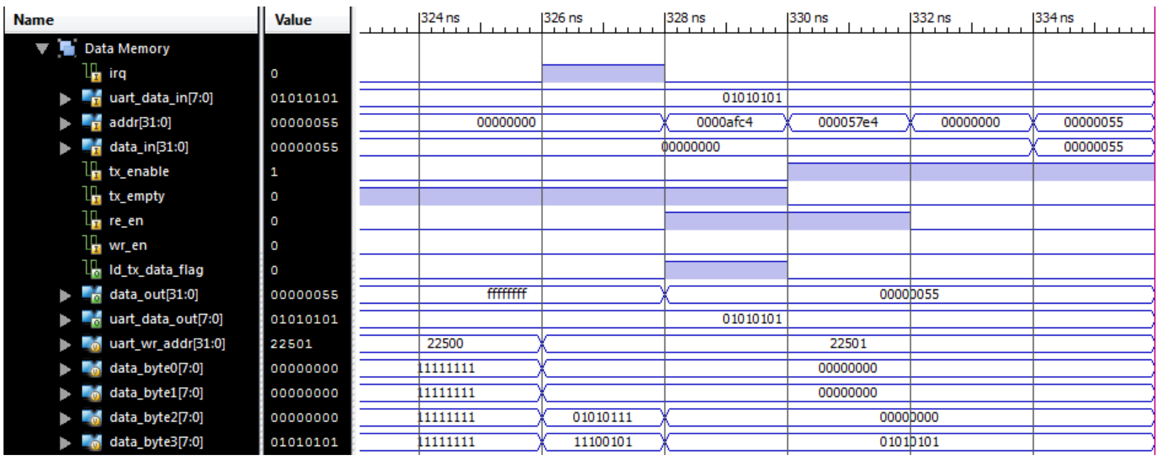


Figure 49. MEM stage inputs and outputs during a UART receive operation.

2. UART Transmit Testing

The testing featured in this subsection focuses on the UART transmit functionality. Transmission on the UART serial interface is accomplished by passing the data written at a specific memory location (address 44999) to the UART output. It was stated in Chapter IV that the data written to this address can only be sent to the UART when a load word instruction references this memory address. This approach has the advantage of avoiding interrupt procedures which delay the processor from completing additional operations on unaffected data.

The transmission data was first placed into the data memory initialization file at address 44999 prior to the start of the simulation. During normal operation of the processor this action is typically performed by the store word instruction which was verified in Section B of this chapter. An additional load word instruction was added to the test program at address 356 of instruction memory and is decoded as 0x8C00AFC4 in

Figure 50. Using the instruction set of Appendix C, we can determine when this instruction attempts to load a word with base register $0 and offset 44996 and store the result back into register $0.

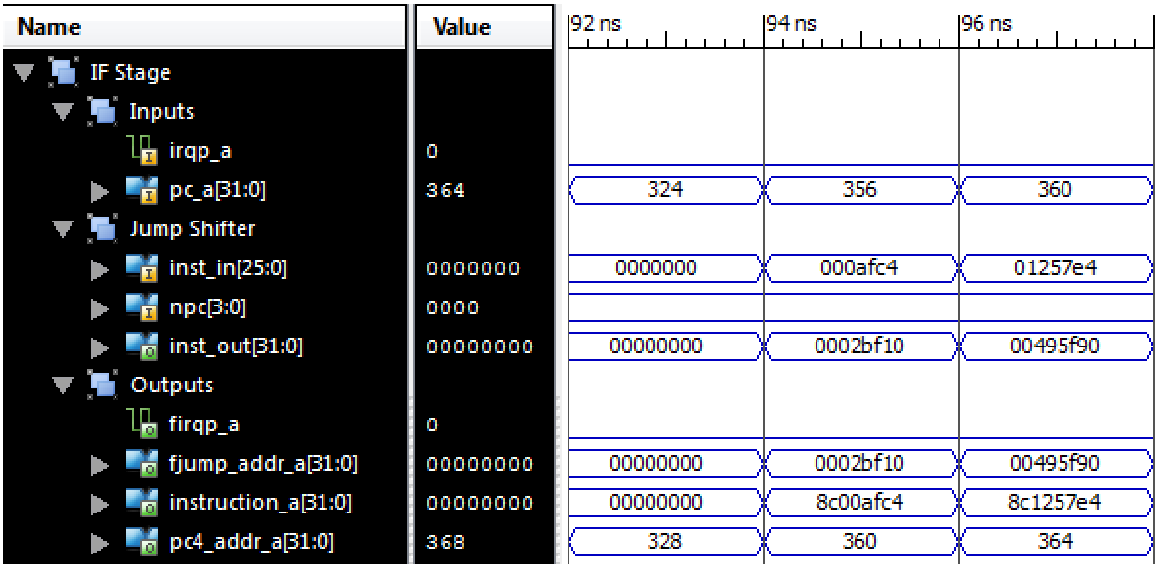


Figure 50. Load word instruction referencing the UART transmit address space.

During the ID and EX stages this particular instruction propagates in the same manner as would any other load word instruction. When the MEM stage is reached, the read address location of 44996 causes the data memory module to interpret this load word instruction as a signal to transmit the data in address 44999 to the UART. The data memory module is programmed to accept any read address input from 44996 – 44999 as referencing the memory mapped to the UART. This activity is shown in Figure 51, where the combination of the **addr** and **re_en** inputs prompts the transition of **uart_data_out** and the **ld_tx_data_flag** bits at 100 ns.

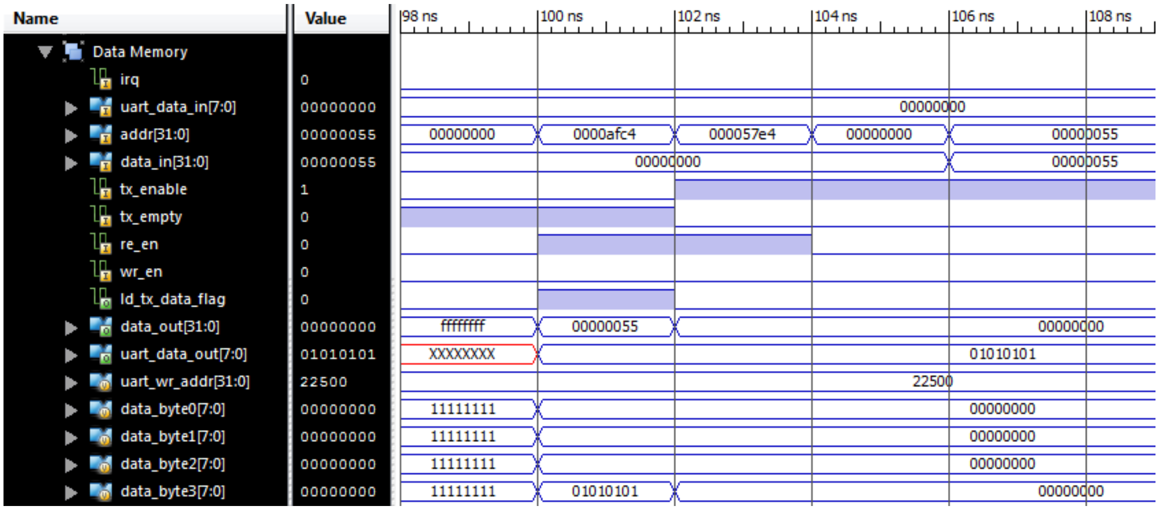


Figure 51. Data memory module inputs and outputs during the UART transmission process.

The **ld_tx_data_flag** is set high when the UART mapped memory is read to provide indication of incoming data to the UART. This data is saved into the **tx_data_a** register, shown in Figure 52, which holds data prior to the **tx_enable_a** flag being set high. This transition occurs in the **tx_data_a** row of Figure 52 at 100 ns. In the next clock cycle beginning at 102 ns, the **tx_empty_a** bit is set low and the **tx_enable_a** bit is set high which indicates data is received in the receive register and transmission begins. Simultaneously, the **tx_out** output of Figure 52, representing the serial transmission medium, transitions from high to low signaling the preamble bit of the UART transmission. This is followed by eight data bits (alternating ones and zeroes) from 102 – 120 ns. At 120 ns of simulation time the **tx_enable_a** bit transitions from high to low and the **tx_empty_a** flag from low to high, indicating the transmission is complete. The **tx_out** bit ends its transmission by transitioning from low to high where it remains until indication of a new transmission is initiated by the **ld_tx_data_flag**.

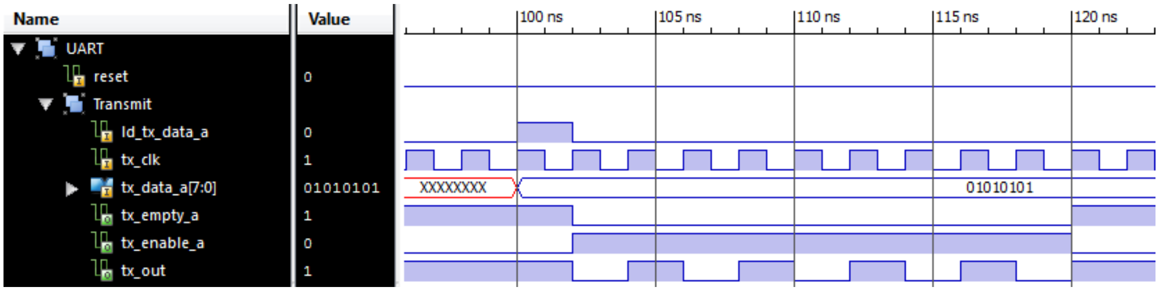


Figure 52. UART signal transitions during a transmission.

D. TROUBLESHOOTING

The majority of components contained within the processor module performed simple tasks and were easy to design using Verilog behavioral modeling. During the testing and verification process, however, some components exhibited unexpected behavior when inserted into the complete processor unit. The most difficult issues encountered in the troubleshooting process were related to the PC selector module. The PC selector module is undoubtedly the most complex component of the processor pipeline since it both interfaces with the external UART and accepts a feedback input from the IF stage. Some of the methods and design principles used to overcome errors when testing the processor are discussed in this section.

Initially, the PC selector module design changes its IRQ return address following the high to low transition of the branch/jump flag and at each update of the PC plus four address while performing ISR instructions. Reassignment of the IRQ return address should occur no more than once during an ISR; during same clock cycle the branch/jump flag transitions from low–to–high. In the ISim software, it was determined during this high–to–low transition of the branch/jump flag that its asserted value was one. This is contrary to the other observed transitions in the simulation, which typically take the value of the final state of the signal. Asserting a value of one during the high–to–low transition of the branch/jump flag combined with the low–to–high transition of the clock resulted in the conditions being met for IRQ return address reassignment. The reason for this unexpected assertion of the branch/jump signal has not yet been determined; however, the issue was resolved by removing assignment of the IRQ return address from the sensitivity list of the primary “always” statement in the Verilog module. A new “always”

statement was constructed that assigns the IRQ return address to the same value of the branch/jump address only at the positive edge of the branch/jump signal.

Another major issue encountered while designing the PC selector module was the assignment of the next PC address during transition from the ISR back to the test program. The PC plus four address, which is received as a feedback input from the IF stage, points to address 45,000 of instruction memory from 320 – 322 ns which is when the ISR to test program transition is imminent. This value is outside the boundaries of instruction memory address space. The PC selector module was designed to recognize this invalid address, reassign the next PC to the IRQ return address, and add four to the current PC holder instead; however, this caused the PC plus four address to have the same value as the current PC register on the following clock cycle. An additional manual reassignment of the current PC register was included for when the processor exits interrupt processing mode and attempts to assign the next PC value. After the second increment of the new PC holder, the values are again synchronized and reassignment can be performed to the PC plus four address received from the IF stage.

The branch–on–less–than–zero (BLZ) and branch–on–less–than–or–equal–to–zero (BLEZ) instructions initially did not work because the ALU recognized these values as unsigned integers. This became apparent in the simulation when the appropriate branch flags were set by the control module in the ID stage, but the comparison bits set by the ALU in the EX stage indicated a value greater than or equal to zero. Thus, the branch multiplexer module did not raise the branch/jump selector flag, and the PC selector module passed the PC plus four address vice the branch/jump address to the current PC register. This issue was mitigated by testing the most significant bit of the first operand to the ALU vice comparing the entire operand to zero. Since negative numbers always have a one as the most significant bit in the two’s complement representation of integers, this provides an equivalent indication of a negative number. Verilog features an integer register declaration to construct registers that are treated as signed vice unsigned values, however, this method was unable to be successfully employed in the payload processor. Several declarations were attempted according to the format provided in [31] but each produced a syntax error in the ISE Webpack compiler. The most significant bit

comparison is adequate for testing operands, but implementing the solution with established Verilog declarations is preferred for future versions of the processor. This prevents the need for additional logic code in the ALU which is required to handle comparisons between registers and perform more complex mathematical operations.

E. CHAPTER SUMMARY

Testing and verification of the payload processor to ensure the correct execution of its entire ISA and interface with the UART was presented in this chapter. First, the pipeline registers were tested to ensure they passed the correct data to the voter circuits. These demonstrations also exhibited the voting logic’s ability to clear SEUs that might occur by selecting the majority of input bits as output. Next, the proper function of the 24 instructions supported in the ISA was summarized by organizing the signals into five groups and tracking their propagation through the pipeline stages. Finally, the processor’s ability to interface with the UART was verified by performing simulated receive and transmit operations.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

The various fault–tolerant methods and hardware on which a CubeSat payload processor could be implemented were investigated, and a design capable of supporting a payload attached by a serial communication link was proposed and tested. While a solid framework for the payload processor was established, a significant amount of testing and development must be accomplished before the device is ready to fly an actual mission. Additionally, various concepts in logic programming development were found to be more beneficial at different levels of processor design. The conclusions section speaks primarily to the logic design methodology, while the future work section outlines several major steps necessary to launch the payload processor as part of a CubeSat mission. Ultimately, the successful development of a payload processor for CubeSat missions can greatly expand the capabilities of the DOD's satellite constellation.

A. CONCLUSIONS

The initial goals proposed in Chapter I of this thesis were successfully accomplished. First, the ITMR fault–tolerant architecture was selected as the best fault–tolerance implementation method for the processor. The Xilinx Virtex–5 FPGA was determined to be the best implementation platform for the processor due to its increased logic resources and performance capabilities over the Actel ProASIC3. A hybrid HDL/schematic design of the processor was then produced along with a UART serving as the serial–to–parallel communication device. Finally, verification of the operation and interaction of these devices was performed using the Xilinx ISim logic simulation software.

The payload processor was a complex and detailed design which required tremendous attention to detail. While building the ITMR pipeline registers a substantial amount of iteration was used in the HDL code. Devices 32 bits or greater in width are efficiently constructed using the Verilog "generate" keyword. This syntax allows the user to create serialized instances of elements using a "for" loop vice declaring and wiring

each individual module in Verilog code or schematics. When the individual pipeline registers were combined into ITMR modules, each of the triplicated members was declared without use of the "generate" keyword. This allowed for better organization of the modules within the ITMR registers, which proved helpful when debugging the processor. Top level synthesis of the complete processor pipeline and combinatorial logic modules within the pipeline stages was performed using schematic files. The schematic layouts allow graphic representation of dataflow within the pipeline and grouping of module input and output pins with regard to their association in the ITMR architecture.

Using a hybrid I/O scheme for the processor offers a flexible approach to memory management. By placing control logic in data memory, the processor avoids an IRQ when sending data to the UART. Expanding the number of flag bits in the processor control module and employing more advanced memory writing techniques can further reduce IRQs. Additionally, the hybrid I/O scheme offers a window into what re–configurability may look like on orbit. While the Virtex–5 block RAM memory primitives in [32] are more restrictive than the simulated RAM used in this thesis, the logic controlling data memory effectively allows re–sizing and re–addressing. This control logic may require an increased level of detail to interface with a Virtex–5 library primitive, however, placement of the memory control logic external to the block RAM will enable the same functionality achieved in this simulation.

B. FUTURE WORK

The payload processor designed in this thesis represents an initial step in the advancement of CubeSat technology. It recycles an effective fault–tolerance method and uses a simple UART interface to communicate with an I/O device. A significant amount of testing and development is required to produce a payload processor able to support a CubeSat mission. The following sections are semi–sequentially ordered to provide a way forward for development.

1. Hazard Detection and Exception Handling

The design proposed in this thesis is not yet capable of handling data and control hazards or exceptions [33] [34]. To take full advantage of the speed and efficiency

pipelined processor offers, hazard detection units, data forwarding modules, and exception handling modules should be developed. Implementation of these modules is relatively complex since they must not only operate in conjunction with the processor, but cooperate with IRQs generated by the UART.

2. Assembler, Compiler, and ISA Expansion

CubeSat designers must be able to use high–level programming languages to instruct the processor on how to interact with and manage its payload. This requires development of an assembler and compiler that are tailored to the processor's ISA. The compiler should support either C or Python as the high–level programming language. Development of the compiler and assembler should be performed concurrently with expansion of the processor ISA to support all instructions in the MIPS core ISA. Further expansion beyond the MIPS core ISA may be required to support onboard processing of data using advanced algorithms.

3. Memory Management

The data and instruction memory currently implemented in the payload processor is small and simplified. The triplicated data memory segments can support a single file no greater than 22.5 kB, and instruction memory limits a program to 11,250 instructions. Additionally, memory management is entirely contained within the data memory module and features no ECCs or caches. Components that handle FPGA memory as a cache or virtual memory system will likely need to be developed allowing for interaction with a secondary, off–chip storage memory. Secondary storage will likely require several gigabytes (GB) of memory for the processor to perform useful operations. The type of secondary memory and ECCs that must be implemented to protect stored bits from SEEs should also be considered in this research.

4. Implementation and Testing

Once the processor hardware design and simulation is finalized, the associated Verilog and schematic files should be implemented and tested on the Virtex–5 as a complete system. Testing will likely require the creation of additional modules that allow

the tester to artificially insert SEUs into the system. Similarly, the desired outputs must be identified and routed to a logic analyzer. The PAR results performed by the logic design software during this stage will also provide insight into the speed of the implemented processor and the quantity of logic resources it consumes on the FPGA. This research should seek to verify the ISim results obtained for the HDL models and define performance parameters for the hardware with the implemented design.

5. Hardware Production

After all of the major hardware components have been designed and successfully tested under a variety of inputs, a PCB with the FPGA and associated I/O connections should be manufactured. This hardware would serve as a flight test model similar to the SADv3 test boards prepared by Parobek for the launch sequencer [1]. The design of a standalone PCB would best be performed in an advanced software environment such as Altium Designer. The completed PCB must be appropriately wired to support all of the desired I/O interfaces, JTAG pins, and power inputs. This PCB would serve the purpose of prototype testing for future CubeSat missions and for radiation testing discussed in the next section.

6. Radiation Testing

As discussed in Chapter III, data pertaining to the radiation hardness of non–RADHARD FPGAs is relatively sparse. It was established in [11] and [35] that as the embedded device under test (DUT) becomes more complex, control over its testing diminishes, and delays resulting from alterations to the original design may overwhelm the project. Since the payload processor is a very complex logic design, comprehensive measurements of the individual logic modules will likely produce the best estimates for overall device radiation hardness. The data obtained in this experiment would provide insight regarding which modules and locations of the FPGA are most susceptible to SEUs. This research would involve coordination with a facility that possesses a cyclotron or other controlled radiation source. Based on the experimental methodology performed in [35], this research should be performed concurrently with that proposed in Subsections 3 and 4 of this section.

C. CLOSING REMARKS

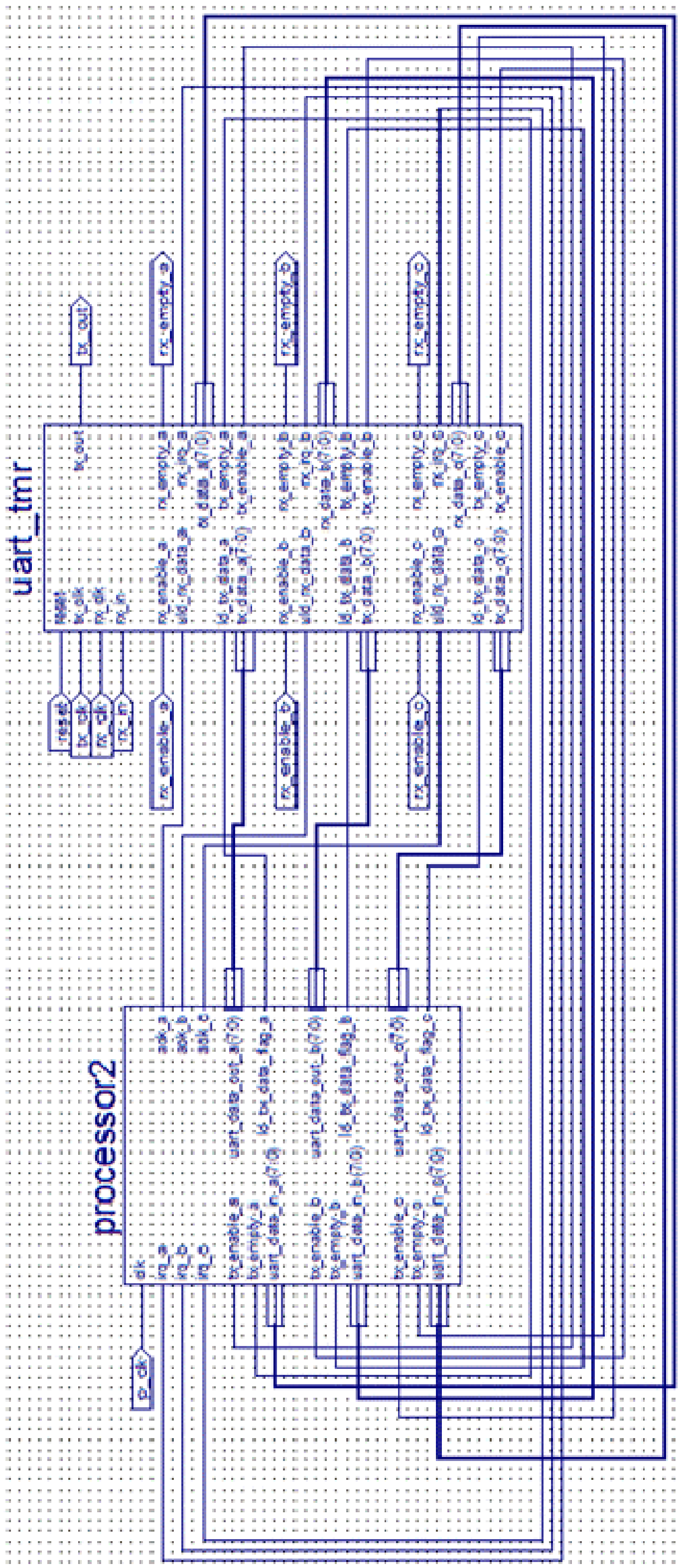
In conclusion, an initial design for the payload processor of a CubeSat featuring a fault–tolerant architecture that prevents specific SEEs from disrupting its operation was proposed and investigated in this thesis. The design contains an ISA of 24 instructions that implement the most fundamental operations of a MIPS processor. Additionally, a generic UART device was proposed and tested concurrently with the processor to ensure the complete system was capable of receiving and transmitting instructions on a serial connection. Though this processor design requires substantial development before it is ready for a mission, it establishes a foundation on which future CubeSat payloads can be developed. Continued research into payload processor design methods and implementation hardware is critical to ensuring the DOD retains a competitive edge in CubeSat technology.

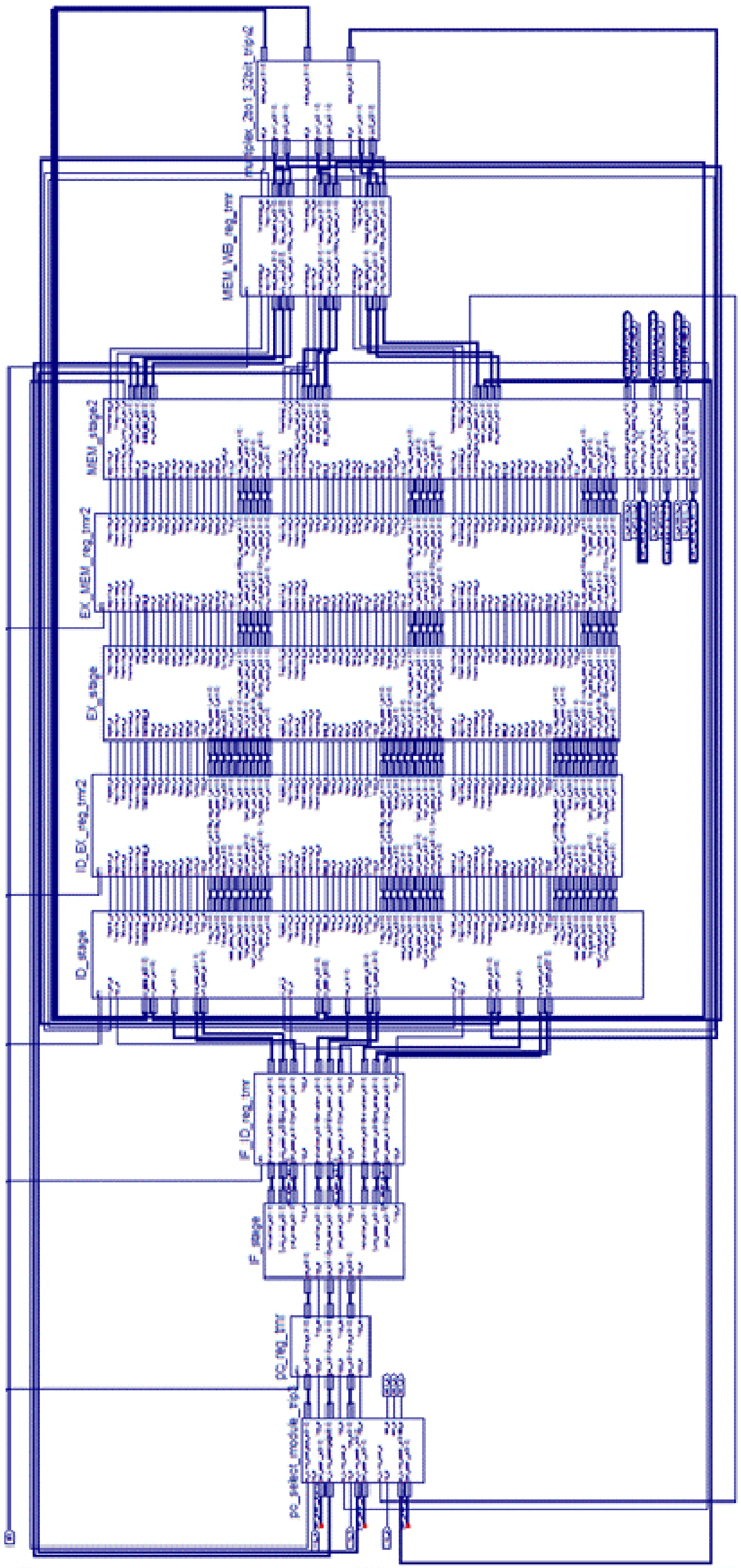
THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. VERILOG CODE AND SCHEMATICS

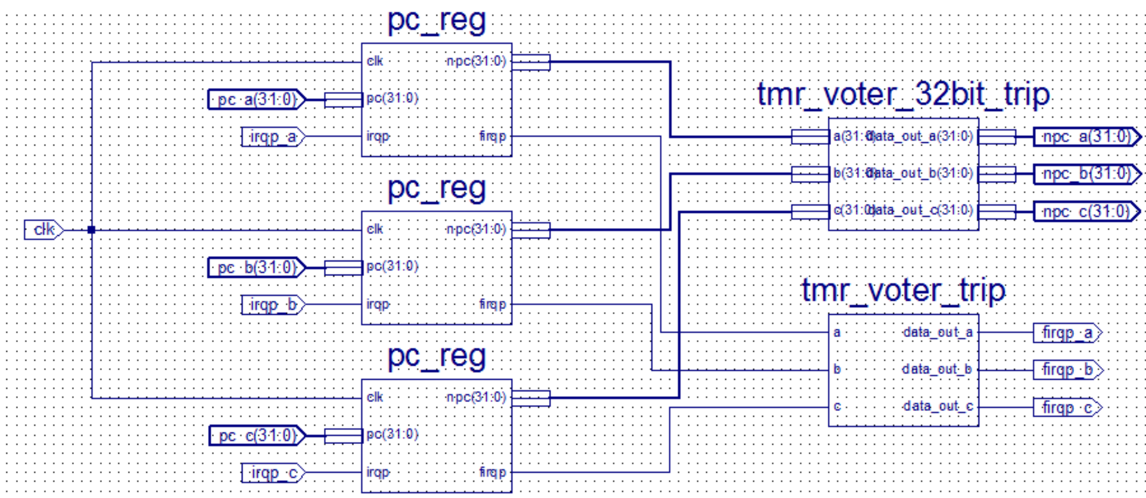
This appendix provides the complete code and schematic files required to build the payload processor model tested in this thesis. Each section consists of a top level pipeline stage or register displayed in a schematic. The components within the pipeline stage schematics are triplicated and consist of a wrapper schematic or Verilog file. Base level components that implement the actual logic functions are primarily Verilog files with some exceptions. The complete payload processor system with the pipeline UART modules followed by the pipeline layout is presented in Section A. Schematics and Verilog files that comprise each stage and pipeline register beginning with the PC register and working sequentially towards the WB stage are displayed in Sections B – K.

A. PAYLOAD PROCESSOR SYSTEM, PIPELINE, AND UART



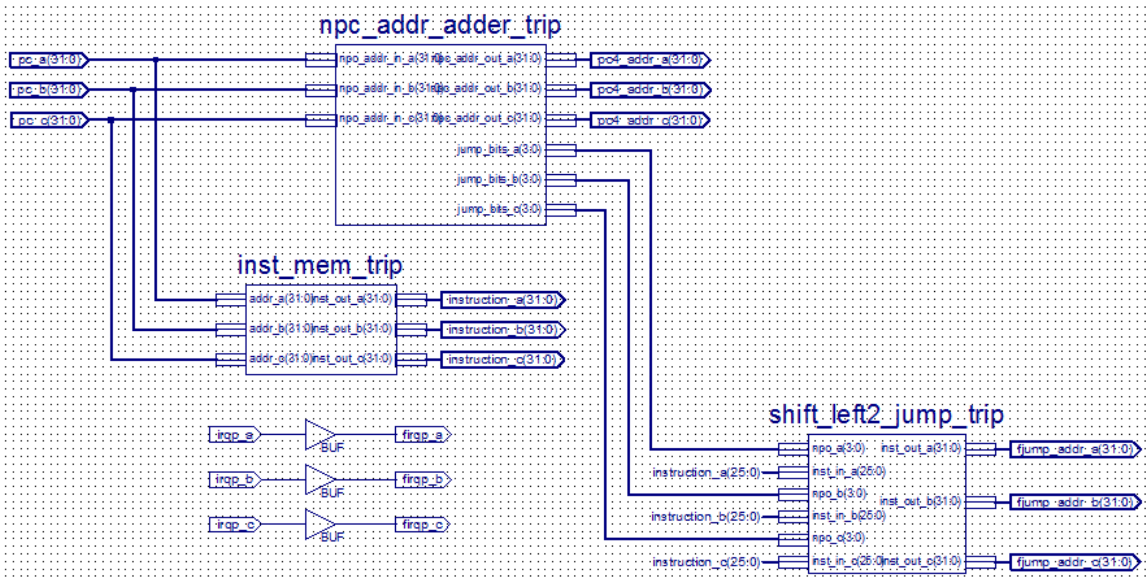

MEM_WB_reg_tmr
MEM_stage2
EX_MEM_reg_tmr2
EX_stage
ID_EX_reg_tmr2
ID_stage
IF_ID_reg_tmr
IF_stage
pc_reg_tmr
pc_select_module_trip1

B. PC REGISTER

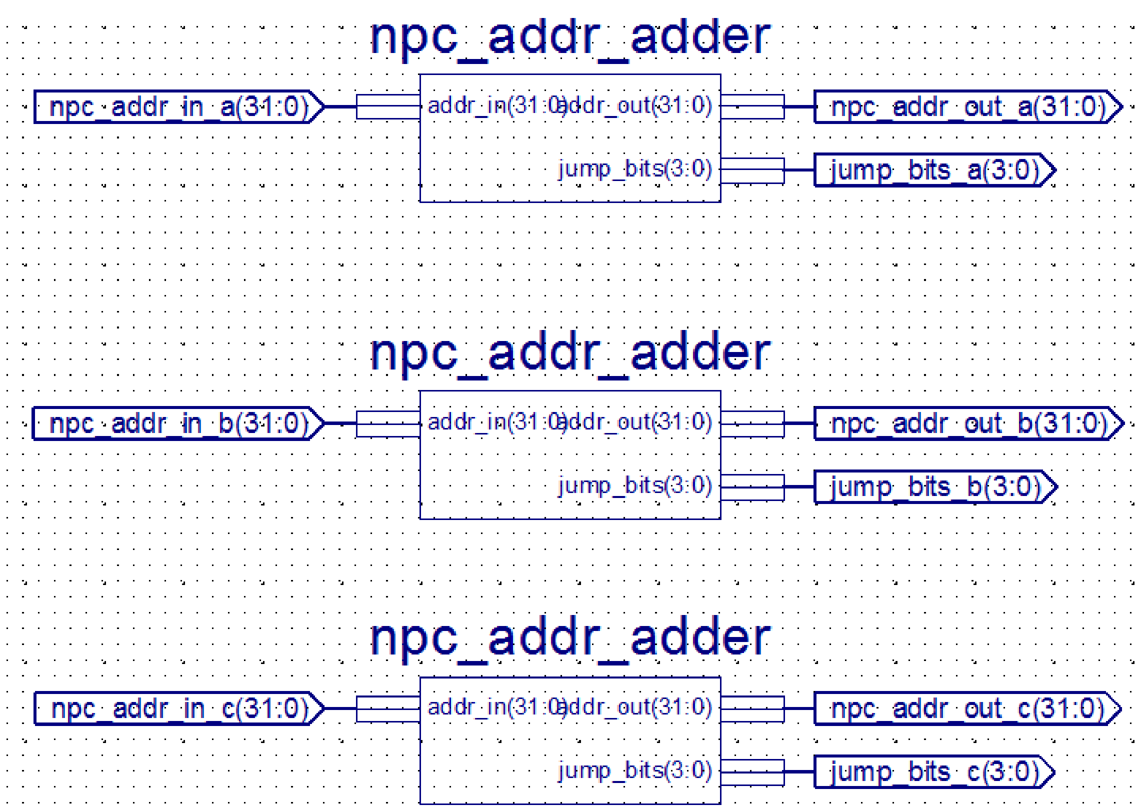



```
module pc_reg(clk, pc, irqp, npc, firqp);

   parameter n = 32; // Processor width

   input [n-1:0] pc; // Input PC value
   input irqp, clk; // Input interrupt request processing flag bit and clock signal
   output reg [n-1:0] npc; // Output PC value
   output reg firqp; // Output interrupt request processing flag bit

   // Swap outputs on the positive edge of the clock
   always @(posedge clk)
      begin
         npc <= pc;
         firqp <= irqp;
      end

   // Initialize values
   initial
      begin
         firqp <= 0;
         npc <= 32'bxxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx;
      end

endmodule
```

C. IF STAGE

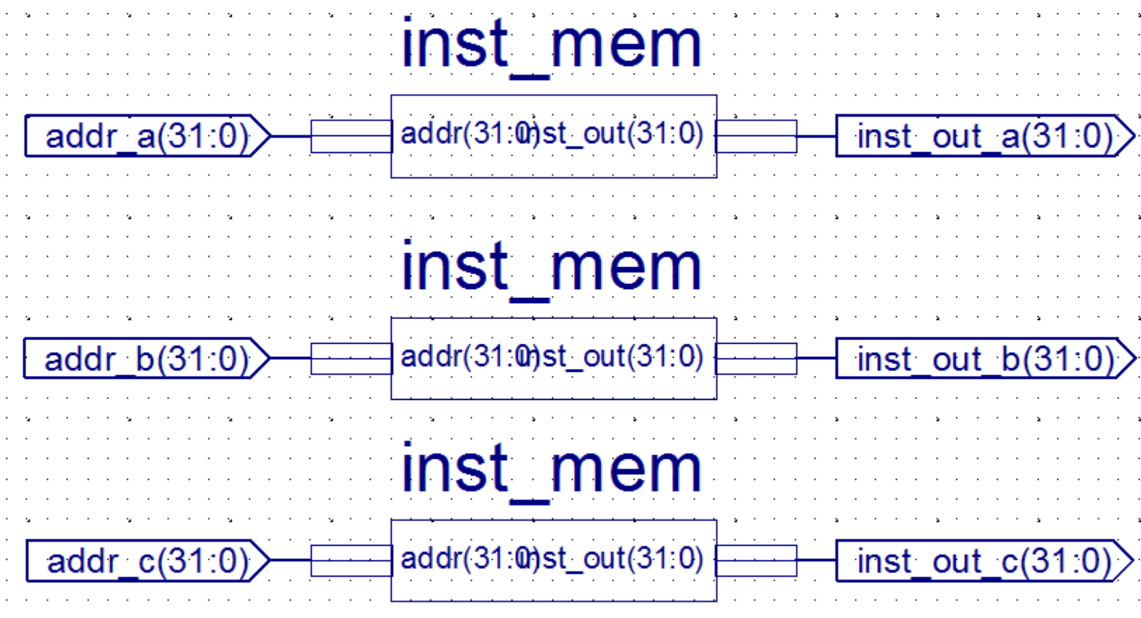


1. New PC Adder Module



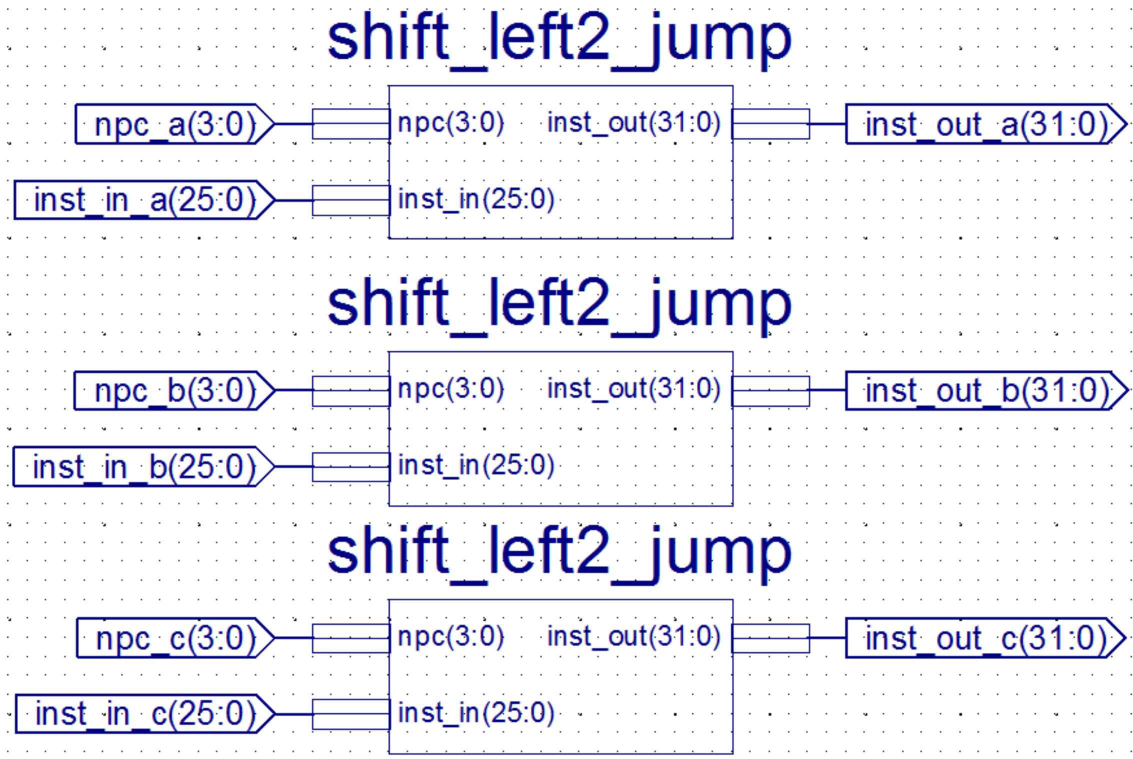
```
module npc_addr_adder(addr_in, addr_out, jump_bits);

   input [31:0] addr_in; // Input address
   output reg [31:0] addr_out; // Output address
   output reg [3:0] jump_bits; // Most significant bits as part of the jump address

   always @ (addr_in)
      addr_out = addr_in + 4; // Calculates the PC + 4 address

   always @ (addr_in)
      jump_bits = addr_in [31:28]; // Forwards the jump address bits

endmodule
```

2. Instruction Memory



```
module inst_mem(addr, inst_out);

   input wire [31:0] addr; // Input address
   output reg [31:0] inst_out; // Output instruction
   reg [7:0] b0; // Byte 0
   reg [7:0] b1; // Byte 1
   reg [7:0] b2; // Byte 2
   reg [7:0] b3; // Byte 3

   reg [7:0] inst_memory_module [0:44999]; // Instruction memory data structure

   // Perform an address to instruction translation each time the input address
   // changes.
   always @(addr)
      begin
         // Each byte is assigned to an offset value based on the start address
         // of the instruction.  Each MIPS instruction is 4 bytes.  Once translated
         // from a memory address to an instruction, the bytes are concatenated.
         b0 = inst_memory_module [addr];
         b1 = inst_memory_module [addr + 1];
         b2 = inst_memory_module [addr + 2];
         b3 = inst_memory_module [addr + 3];
         inst_out = {b0, b1, b2, b3};
      end

   // An initial file is provided with the instructions already translated into
   // machine language.
   initial
      begin
         $readmemb ("payload_processor_inst_memory_init5.txt", inst_memory_module);
         inst_out = 0;
      end

endmodule
```

3. Jump Address Calculator

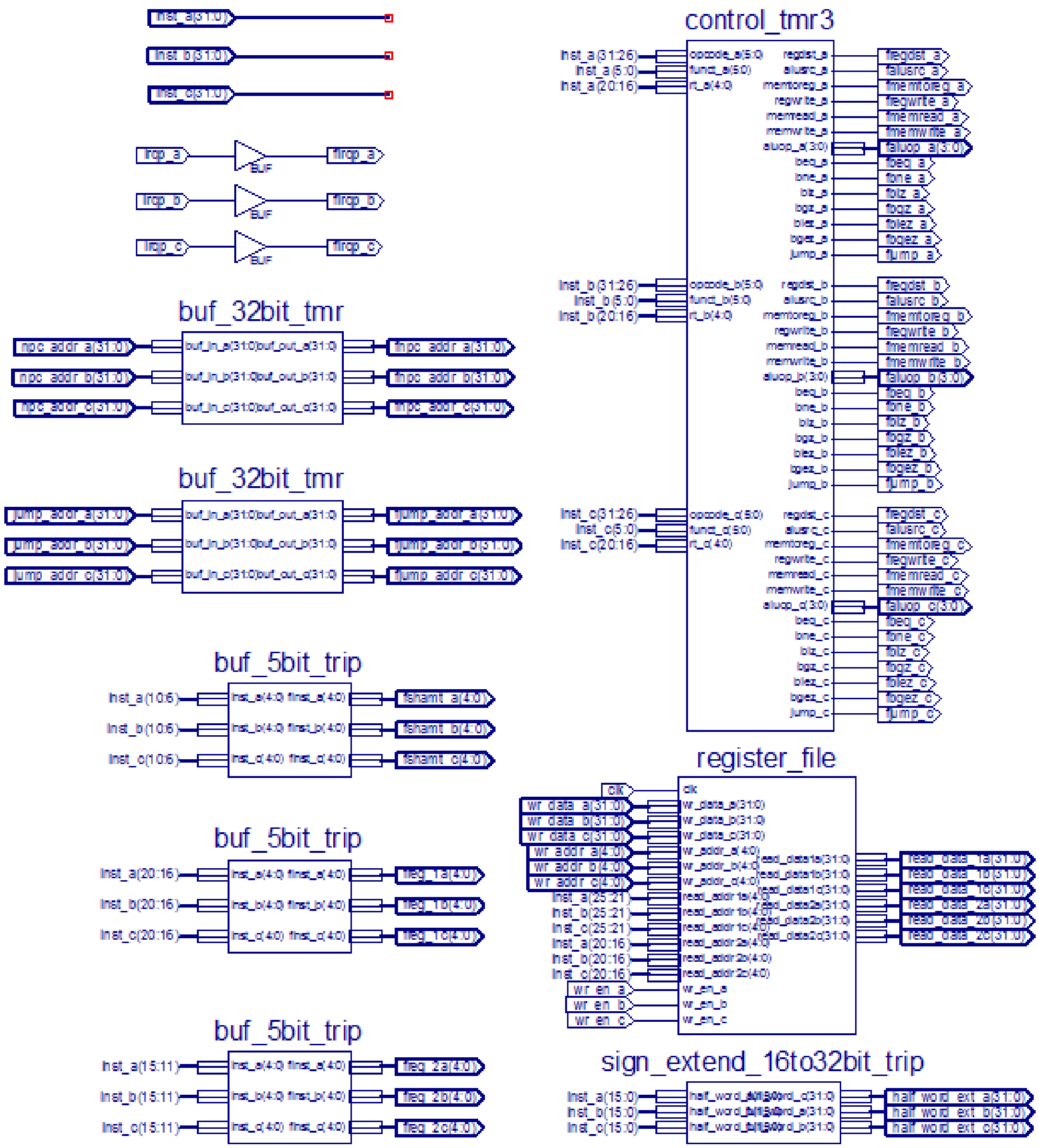



```
module shift_left2_jump(npc, inst_in, inst_out);

   input [25:0] inst_in; // Number of words to jump in instruction memory
   input [3:0] npc; // Four most significant bits from PC
   output reg [31:0] inst_out; // Output instruction address

   initial
      inst_out = 0;

   // At each transition in the instruction address or PC bits, concatenate the
   // PC bits, jump offset, and two zeroes.  The appended zeroes ensure the next
   // register address will be the start of a word.
   always @ (inst_in or npc)
      inst_out = {npc, inst_in, 2'b00};


endmodule
```

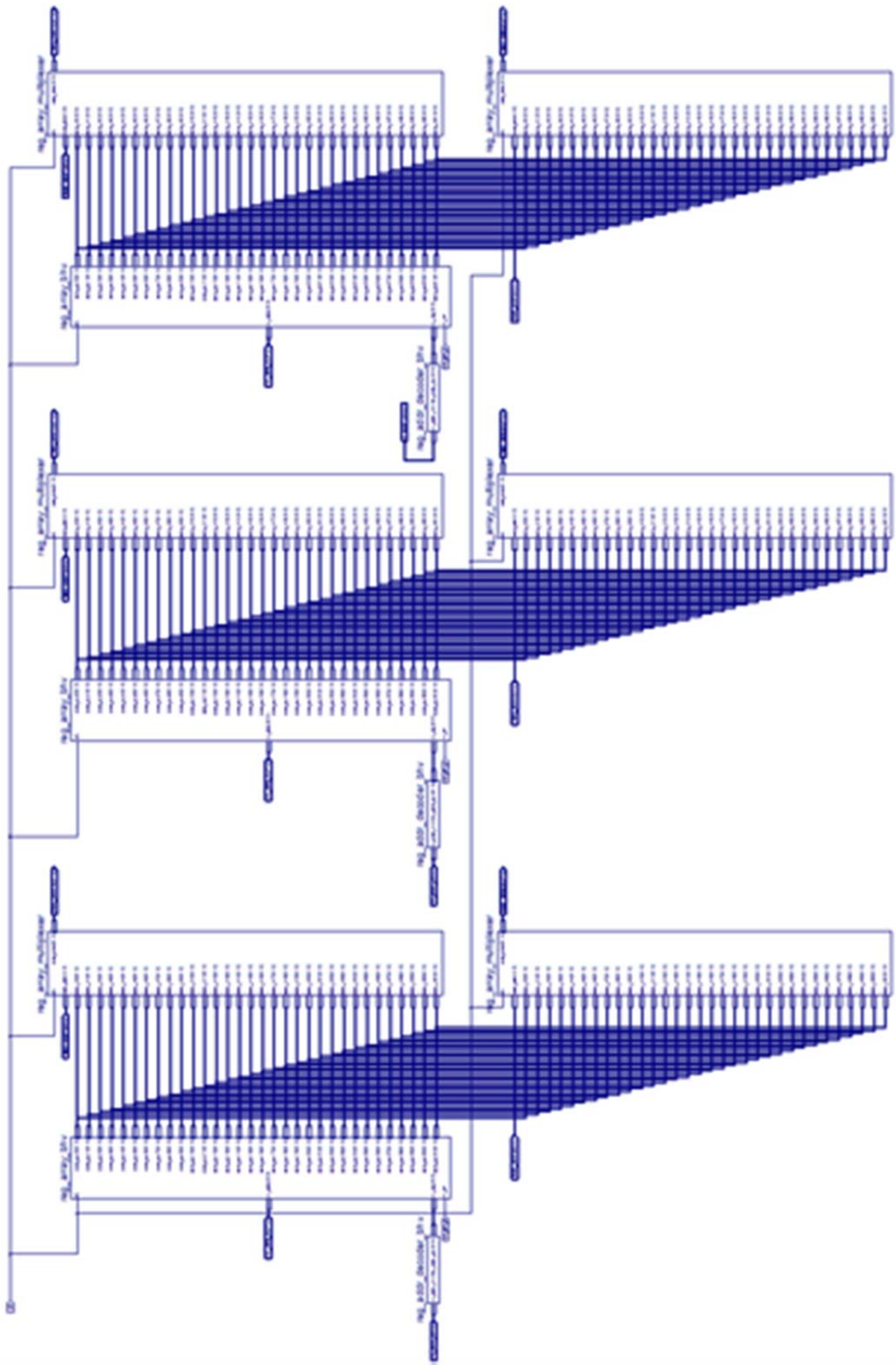
D. IF/ID REGISTER

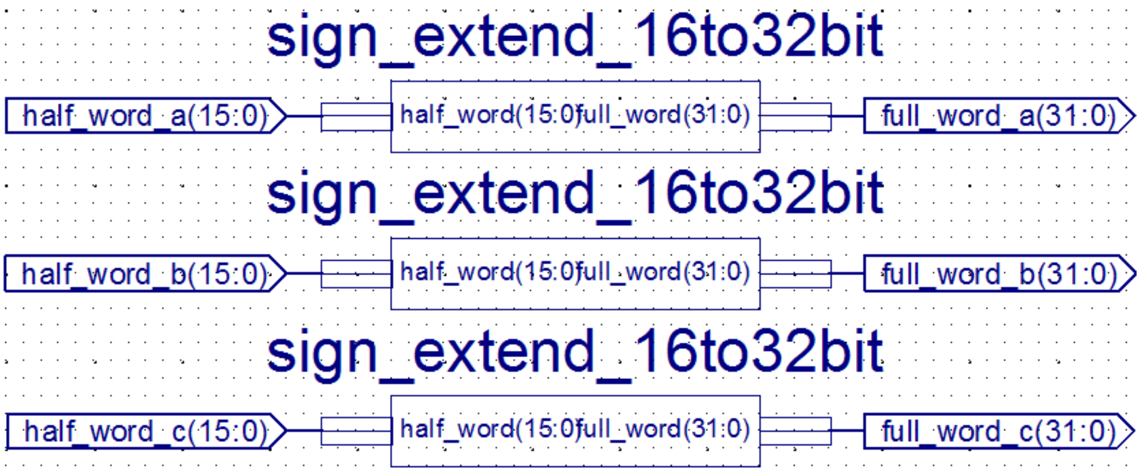
```
module IF_ID_reg_tmr(clk, instruction_a, pc4_addr_a, jump_addr_a, irqp_a,
   finstruction_a, fpc4_addr_a, fjump_addr_a, firqp_a, instruction_b, pc4_addr_b,
   jump_addr_b, irqp_b, finstruction_b, fpc4_addr_b, fjump_addr_b, firqp_b,
   instruction_c, pc4_addr_c, jump_addr_c, irqp_c, finstruction_c, fpc4_addr_c,
   fjump_addr_c, firqp_c);

   // Inputs
   input clk; // Clock signal
   input irqp_a, irqp_b, irqp_c; // Interrupt request processing bits
   input [31:0] instruction_a, instruction_b, instruction_c; // Instructions
   input [31:0] pc4_addr_a, pc4_addr_b, pc4_addr_c; // PC + 4 addresses
   input [31:0] jump_addr_a, jump_addr_b, jump_addr_c; // Jump addresses

   // Wires connecting the registers to voters
   wire nirqp_a, nirqp_b, nirqp_c;
   wire [31:0] ninstruction_a, ninstruction_b, ninstruction_c;
   wire [31:0] npc4_addr_a, npc4_addr_b, npc4_addr_c;
   wire [31:0] njump_addr_a, njump_addr_b, njump_addr_c;

   // Outputs
   output firqp_a, firqp_b, firqp_c;
   output [31:0] finstruction_a, finstruction_b, finstruction_c;
   output [31:0] fpc4_addr_a, fpc4_addr_b, fpc4_addr_c;
   output [31:0] fjump_addr_a, fjump_addr_b, fjump_addr_c;

   // TMR voters
   tmr_voter_32bit_trip voter0 (.a(ninstruction_a), .b(ninstruction_b), .c(ninstruction_c),
   .data_out_a(finstruction_a), .data_out_b(finstruction_b), .data_out_c(finstruction_c));
   tmr_voter_32bit_trip voter1 (.a(npc4_addr_a), .b(npc4_addr_b), .c(npc4_addr_c),
   .data_out_a(fpc4_addr_a), .data_out_b(fpc4_addr_b), .data_out_c(fpc4_addr_c));
   tmr_voter_32bit_trip voter2 (.a(njump_addr_a), .b(njump_addr_b), .c(njump_addr_c),
   .data_out_a(fjump_addr_a), .data_out_b(fjump_addr_b), .data_out_c(fjump_addr_c));
   tmr_voter_trip voter3 (.a(nirqp_a), .b(nirqp_b), .c(nirqp_c), .data_out_a(firqp_a),
   .data_out_b(firqp_b), .data_out_c(firqp_c));

   // Pipeline registers
   IF_ID_reg reg_a (.clk(clk), .instruction(instruction_a), .pc4_addr(pc4_addr_a),
   .jump_addr(jump_addr_a), .irqp(irqp_a), .finstruction(ninstruction_a),
   .fpc4_addr(npc4_addr_a), .fjump_addr(njump_addr_a), .firqp(nirqp_a));
   IF_ID_reg reg_b (.clk(clk), .instruction(instruction_b), .pc4_addr(pc4_addr_b),
   .jump_addr(jump_addr_b), .irqp(irqp_b), .finstruction(ninstruction_b),
   .fpc4_addr(npc4_addr_b), .fjump_addr(njump_addr_b), .firqp(nirqp_b));
   IF_ID_reg reg_c (.clk(clk), .instruction(instruction_c), .pc4_addr(pc4_addr_c),
   .jump_addr(jump_addr_c), .irqp(irqp_c), .finstruction(ninstruction_c),
   .fpc4_addr(npc4_addr_c), .fjump_addr(njump_addr_c), .firqp(nirqp_c));

endmodule
```

E. ID STAGE


control_tmr3
buf_32bit_tmr
buf_32bit_tmr
buf_5bit_trip
buf_5bit_trip
buf_5bit_trip
register_file
sign_extend_16to32bit_trip

```
module control_tmr3(opcode_a, funct_a, rt_a, regdst_a, alusrc_a, memtoreg_a,
   regwrite_a, memread_a, memwrite_a, beq_a, bne_a, blz_a, bgz_a, blez_a, bgez_a,
   jump_a, aluop_a, opcode_b, funct_b, rt_b, regdst_b, alusrc_b, memtoreg_b,
   regwrite_b, memread_b, memwrite_b, beq_b, bne_b, blz_b, bgz_b, blez_b, bgez_b,
   jump_b, aluop_b, opcode_c, funct_c, rt_c, regdst_c, alusrc_c, memtoreg_c,
   regwrite_c, memread_c, memwrite_c, beq_c, bne_c, blz_c, bgz_c, blez_c, bgez_c,
   jump_c, aluop_c);

   input [5:0] opcode_a, opcode_b, opcode_c;
   input [5:0] funct_a, funct_b, funct_c;
   input [4:0] rt_a, rt_b, rt_c;

   wire nregdst_a, nregdst_b, nregdst_c;
   wire nalusrc_a, nalusrc_b, nalusrc_c;
   wire nmemtoreg_a, nmemtoreg_b, nmemtoreg_c;
   wire nregwrite_a, nregwrite_b, nregwrite_c;
   wire nmemread_a, nmemread_b, nmemread_c;
   wire nmemwrite_a, nmemwrite_b, nmemwrite_c;
   wire nbeq_a, nbeq_b, nbeq_c;
   wire nbne_a, nbne_b, nbne_c;
   wire nblz_a, nblz_b, nblz_c;
   wire nbgz_a, nbgz_b, nbgz_c;
   wire nblez_a, nblez_b, nblez_c;
   wire nbgez_a, nbgez_b, nbgez_c;
   wire njump_a, njump_b, njump_c;
   wire [3:0] naluop_a, naluop_b, naluop_c;

   output regdst_a, regdst_b, regdst_c;
   output alusrc_a, alusrc_b, alusrc_c;
   output memtoreg_a, memtoreg_b, memtoreg_c;
   output regwrite_a, regwrite_b, regwrite_c;
   output memread_a, memread_b, memread_c;
   output memwrite_a, memwrite_b, memwrite_c;
   output beq_a, beq_b, beq_c;
   output bne_a, bne_b, bne_c;
   output blz_a, blz_b, blz_c;
   output bgz_a, bgz_b, bgz_c;
   output blez_a, blez_b, blez_c;
   output bgez_a, bgez_b, bgez_c;
   output jump_a, jump_b, jump_c;
   output [3:0] aluop_a, aluop_b, aluop_c;

   control cnt_a (.opcode(opcode_a), .funct(funct_a), .rt(rt_a), .regdst(regdst_a),
   .alusrc(alusrc_a), .memtoreg(memtoreg_a), .regwrite(regwrite_a), .memread(memread_a),
   .memwrite(memwrite_a), .beq(beq_a), .bne(bne_a), .blz(blz_a), .bgz(bgz_a),
   .blez(blez_a), .bgez(bgez_a), .jump(jump_a), .aluop(aluop_a));

   control cnt_b (.opcode(opcode_b), .funct(funct_b), .rt(rt_b), .regdst(regdst_b),
   .alusrc(alusrc_b), .memtoreg(memtoreg_b), .regwrite(regwrite_b), .memread(memread_b),
   .memwrite(memwrite_b), .beq(beq_b), .bne(bne_b), .blz(blz_b), .bgz(bgz_b),
   .blez(blez_b), .bgez(bgez_b), .jump(jump_b), .aluop(aluop_b));

   control cnt_c (.opcode(opcode_c), .funct(funct_c), .rt(rt_c), .regdst(regdst_c),
   .alusrc(alusrc_c), .memtoreg(memtoreg_c), .regwrite(regwrite_c), .memread(memread_c),
   .memwrite(memwrite_c), .beq(beq_c), .bne(bne_c), .blz(blz_c), .bgz(bgz_c),
   .blez(blez_c), .bgez(bgez_c), .jump(jump_c), .aluop(aluop_c));

endmodule
```





1. Control Module

```
module control(opcode, funct, rt, regdst, alusrc, memtoreg, regwrite, memread, memwrite,
   beq, bne, blz, bgz, blez, bgez, jump, aluop);

   input [5:0] opcode;
   input [5:0] funct;
   input [4:0] rt;

   output reg regdst, alusrc, memtoreg, regwrite, memread, memwrite;
   output reg beq, bne, blz, bgz, blez, bgez, jump;
   output reg [3:0] aluop;

   initial
      begin
         regdst <= 0;
         alusrc <= 0;
         memtoreg <= 0;
         regwrite <= 0;
         memread <= 0;
         memwrite <= 0;
         beq <= 0;
         bne <= 0;
         blz <= 0;
         bgz <= 0;
         blez <= 0;
         bgez <= 0;
         jump <= 0;
         aluop <= 0;
      end
```
(continued on next page)

```
always @ (opcode or funct or rt)
   begin
      if (opcode == 0)
         begin
            // ADD, ADDU (Add, Add Unsigned)
            // R-Format
            if (funct == 32 | funct == 33)
               begin
                  regdst = 1'b1;
                  alusrc = 1'b0;
                  memtoreg = 1'b0;
                  regwrite = 1'b1;
                  memread = 1'b0;
                  memwrite = 1'b0;
                  jump = 1'b0;
                  beq = 1'b0;
                  bne = 1'b0;
                  blz = 1'b0;
                  bgz = 1'b0;
                  blez = 1'b0;
                  bgez = 1'b0;
                  aluop = 0;
               end

            // SUB, SUBU (Subtract, Subtract Unsigned)
            else if (funct == 34 | funct == 35)
               begin
                  regdst = 1'b1;
                  alusrc = 1'b0;
                  memtoreg = 1'b0;
                  regwrite = 1'b1;
                  memread = 1'b0;
                  memwrite = 1'b0;
                  jump = 1'b0;
                  beq = 1'b0;
                  bne = 1'b0;
                  blz = 1'b0;
                  bgz = 1'b0;
                  blez = 1'b0;
                  bgez = 1'b0;
                  aluop = 1;
               end

            // AND (Logical)
            else if (funct == 36)
               begin
                  regdst = 1'b1;
                  alusrc = 1'b0;
                  memtoreg = 1'b0;
                  regwrite = 1'b1;
                  memread = 1'b0;
                  memwrite = 1'b0;
                  jump = 1'b0;
                  beq = 1'b0;
                  bne = 1'b0;
                  blz = 1'b0;
                  bgz = 1'b0;
                  blez = 1'b0;
                  bgez = 1'b0;
                  aluop = 2;
               end
```

(continued on next page)

```
// OR (Logical)
else if (funct == 37)
   begin
      regdst = 1'b1;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 3;
   end

// XOR (Logical)
else if (funct == 38)
   begin
      regdst = 1'b1;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 4;
   end

// NOR (Logical)
else if (funct == 39)
   begin
      regdst = 1'b1;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 5;
   end
```

(continued on next page)

```
// SLT (Set on Less than)
else if (funct == 42)
   begin
      regdst = 1'b1;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 10;
   end

// SLLV (Shift Left Logical Variable)
else if (funct == 4)
   begin
      regdst = 1'b1;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 6;
   end

// SLL (Shift Left Logical)
else if (funct == 0)
   begin
      regdst = 1'b1;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 7;
   end
```

(continued on next page)

```
            // SRLV (Shift Right Logical Variable)
            else if (funct == 6)
               begin
                  regdst = 1'b1;
                  alusrc = 1'b0;
                  memtoreg = 1'b0;
                  regwrite = 1'b1;
                  memread = 1'b0;
                  memwrite = 1'b0;
                  jump = 1'b0;
                  beq = 1'b0;
                  bne = 1'b0;
                  blz = 1'b0;
                  bgz = 1'b0;
                  blez = 1'b0;
                  bgez = 1'b0;
                  aluop = 8;
               end

            // SRL (Shift Right Logical)
            else if (funct == 2)
               begin
                  regdst = 1'b1;
                  alusrc = 1'b0;
                  memtoreg = 1'b0;
                  regwrite = 1'b1;
                  memread = 1'b0;
                  memwrite = 1'b0;
                  jump = 1'b0;
                  beq = 1'b0;
                  bne = 1'b0;
                  blz = 1'b0;
                  bgz = 1'b0;
                  blez = 1'b0;
                  bgez = 1'b0;
                  aluop = 9;
               end
         end

      // ADDI, ADDIU (Add Immediate, Unsigned)
      else if (opcode == 8 | opcode == 9)
         begin
            regdst = 1'b0;
            alusrc = 1'b1;
            memtoreg = 1'b0;
            regwrite = 1'b1;
            memread = 1'b0;
            memwrite = 1'b0;
            jump = 1'b0;
            beq = 1'b0;
            bne = 1'b0;
            blz = 1'b0;
            bgz = 1'b0;
            blez = 1'b0;
            bgez = 1'b0;
            aluop = 0;
         end
```

(continued on next page)

```verilog
// ANDI (Logical And Immediate)
else if (opcode == 12)
   begin
      regdst = 1'b0;
      alusrc = 1'b1;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 2;
   end

// ORI (Logical Or Immediate)
else if (opcode == 13)
   begin
      regdst = 1'b0;
      alusrc = 1'b1;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 3;
   end

// SLTI (Set on Less than Immediate)
else if (opcode == 10)
   begin
      regdst = 1'b0;
      alusrc = 1'b1;
      memtoreg = 1'b0;
      regwrite = 1'b1;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 10;
   end
```

(continued on next page)

```
// BEQ (Branch on Equal)
else if (opcode == 4)
   begin
      regdst = 1'b0;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b0;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b1;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 0;
   end

// BNE (Branch on Not Equal)
else if (opcode == 5)
   begin
      regdst = 1'b0;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b0;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b1;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 0;
   end

// BGEZ (Branch on Greater than or Equal to Zero)
else if (opcode == 1 & rt == 1)
   begin
      regdst = 1'b0;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b0;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b1;
      aluop = 0;
   end
```

(continued on next page)

```verilog
// BLEZ (Branch on Less than or Equal to Zero)
else if (opcode == 6 & rt == 0)
   begin
      regdst = 1'b0;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b0;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b0;
      blez = 1'b1;
      bgez = 1'b0;
      aluop = 0;
   end

// BGTZ (Branch on Greater than Zero)
else if (opcode == 7 & rt == 0)
   begin
      regdst = 1'b0;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b0;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b0;
      bgz = 1'b1;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 0;
   end

// BLTZ (Branch on Less than Zero)
else if (opcode == 1 & rt == 0)
   begin
      regdst = 1'b0;
      alusrc = 1'b0;
      memtoreg = 1'b0;
      regwrite = 1'b0;
      memread = 1'b0;
      memwrite = 1'b0;
      jump = 1'b0;
      beq = 1'b0;
      bne = 1'b0;
      blz = 1'b1;
      bgz = 1'b0;
      blez = 1'b0;
      bgez = 1'b0;
      aluop = 0;
   end
```

(continued on next page)

```
        // J (Jump)
        else if (opcode == 2)
           begin
              regdst = 1'b0;
              alusrc = 1'b0;
              memtoreg = 1'b0;
              regwrite = 1'b0;
              memread = 1'b0;
              memwrite = 1'b0;
              jump = 1'b1;
              beq = 1'b0;
              bne = 1'b0;
              blz = 1'b0;
              bgz = 1'b0;
              blez = 1'b0;
              bgez = 1'b0;
              aluop = 0;
           end

        // LW (Load Word)
        else if (opcode == 35)
           begin
              regdst = 1'b0;
              alusrc = 1'b1;
              memtoreg = 1'b1;
              regwrite = 1'b1;
              memread = 1'b1;
              memwrite = 1'b0;
              jump = 1'b0;
              beq = 1'b0;
              bne = 1'b0;
              blz = 1'b0;
              bgz = 1'b0;
              blez = 1'b0;
              bgez = 1'b0;
              aluop = 0;
           end

        // SW (Store Word)
        else if (opcode == 43)
           begin
              regdst = 1'b0;
              alusrc = 1'b1;
              memtoreg = 1'b0;
              regwrite = 1'b0;
              memread = 1'b0;
              memwrite = 1'b1;
              jump = 1'b0;
              beq = 1'b0;
              bne = 1'b0;
              blz = 1'b0;
              bgz = 1'b0;
              blez = 1'b0;
              bgez = 1'b0;
              aluop = 0;
           end
     end

endmodule
```

2. Register File

```
module reg_addr_decoder_bhv(wr_addr_in, wr_addr_out);
   input [4:0] wr_addr_in; // Input write address
   output reg [31:0] wr_addr_out; // Output write address

   // Initialize values
   initial
      wr_addr_out = 0;

   // For each transition of the input write address, decode the 5-bit write address
   // into 32 individual write enable signals.  Each of the 32 bits is routed to
   // one register in the register file as a write enable signal
   always @(wr_addr_in)
      begin
         if(wr_addr_in == 0)
            wr_addr_out = 32'b00000000_00000000_00000000_00000001;
         else if(wr_addr_in == 1)
            wr_addr_out = 32'b00000000_00000000_00000000_00000010;
         else if(wr_addr_in == 2)
            wr_addr_out = 32'b00000000_00000000_00000000_00000100;
         else if(wr_addr_in == 3)
            wr_addr_out = 32'b00000000_00000000_00000000_00001000;
         else if(wr_addr_in == 4)
            wr_addr_out = 32'b00000000_00000000_00000000_00010000;
         else if(wr_addr_in == 5)
            wr_addr_out = 32'b00000000_00000000_00000000_00100000;
         else if(wr_addr_in == 6)
            wr_addr_out = 32'b00000000_00000000_00000000_01000000;
         else if(wr_addr_in == 7)
            wr_addr_out = 32'b00000000_00000000_00000000_10000000;
         else if(wr_addr_in == 8)
            wr_addr_out = 32'b00000000_00000000_00000001_00000000;
         else if(wr_addr_in == 9)
            wr_addr_out = 32'b00000000_00000000_00000010_00000000;
         else if(wr_addr_in == 10)
            wr_addr_out = 32'b00000000_00000000_00000100_00000000;
         else if(wr_addr_in == 11)
            wr_addr_out = 32'b00000000_00000000_00001000_00000000;
         else if(wr_addr_in == 12)
            wr_addr_out = 32'b00000000_00000000_00010000_00000000;
         else if(wr_addr_in == 13)
            wr_addr_out = 32'b00000000_00000000_00100000_00000000;
         else if(wr_addr_in == 14)
            wr_addr_out = 32'b00000000_00000000_01000000_00000000;
         else if(wr_addr_in == 15)
            wr_addr_out = 32'b00000000_00000000_10000000_00000000;
         else if(wr_addr_in == 16)
            wr_addr_out = 32'b00000000_00000001_00000000_00000000;
         else if(wr_addr_in == 17)
            wr_addr_out = 32'b00000000_00000010_00000000_00000000;
         else if(wr_addr_in == 18)
            wr_addr_out = 32'b00000000_00000100_00000000_00000000;
         else if(wr_addr_in == 19)
            wr_addr_out = 32'b00000000_00001000_00000000_00000000;
         else if(wr_addr_in == 20)
            wr_addr_out = 32'b00000000_00010000_00000000_00000000;
         else if(wr_addr_in == 21)
            wr_addr_out = 32'b00000000_00100000_00000000_00000000;
         else if(wr_addr_in == 22)
            wr_addr_out = 32'b00000000_01000000_00000000_00000000;
         else if(wr_addr_in == 23)
            wr_addr_out = 32'b00000000_10000000_00000000_00000000;
```

(continued on next page)

```
        else if(wr_addr_in == 24)
           wr_addr_out = 32'b00000001_00000000_00000000_00000000;
        else if(wr_addr_in == 25)
           wr_addr_out = 32'b00000010_00000000_00000000_00000000;
        else if(wr_addr_in == 26)
           wr_addr_out = 32'b00000100_00000000_00000000_00000000;
        else if(wr_addr_in == 27)
           wr_addr_out = 32'b00001000_00000000_00000000_00000000;
        else if(wr_addr_in == 28)
           wr_addr_out = 32'b00010000_00000000_00000000_00000000;
        else if(wr_addr_in == 29)
           wr_addr_out = 32'b00100000_00000000_00000000_00000000;
        else if(wr_addr_in == 30)
           wr_addr_out = 32'b01000000_00000000_00000000_00000000;
        else if(wr_addr_in == 31)
           wr_addr_out = 32'b10000000_00000000_00000000_00000000;
     end

endmodule


module reg_array_bhv (wr_reg, wr_en, wr_data, clk, data_out0, data_out1, data_out2,
   data_out3, data_out4, data_out5, data_out6, data_out7, data_out8, data_out9,
   data_out10, data_out11, data_out12, data_out13, data_out14, data_out15,
   data_out16, data_out17, data_out18, data_out19, data_out20, data_out21,
   data_out22, data_out23, data_out24, data_out25, data_out26, data_out27,
   data_out28, data_out29, data_out30, data_out31);

   input [31:0] wr_data, wr_reg; // Write data and write enable signals
   input clk, wr_en; // wr_en is the forwarded regwrite flag from the control module
   // Output wires for 32 registers
   output wire [31:0] data_out0, data_out1, data_out2, data_out3, data_out4;
   output wire [31:0] data_out5, data_out6, data_out7, data_out8, data_out9;
   output wire [31:0] data_out10, data_out11, data_out12, data_out13, data_out14;
   output wire [31:0] data_out15, data_out16, data_out17, data_out18, data_out19;
   output wire [31:0] data_out20, data_out21, data_out22, data_out23, data_out24;
   output wire [31:0] data_out25, data_out26, data_out27, data_out28, data_out29;
   output wire [31:0] data_out30, data_out31;

   // 32 behavioral registers, each receiving the data, write enable, and clock signals
   // in addition to its decoded address bit.
   reg_bhv reg0 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[0]), .clk(clk), .Q(data_out0));
   reg_bhv reg1 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[1]), .clk(clk), .Q(data_out1));
   reg_bhv reg2 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[2]), .clk(clk), .Q(data_out2));
   reg_bhv reg3 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[3]), .clk(clk), .Q(data_out3));
   reg_bhv reg4 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[4]), .clk(clk), .Q(data_out4));
   reg_bhv reg5 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[5]), .clk(clk), .Q(data_out5));
   reg_bhv reg6 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[6]), .clk(clk), .Q(data_out6));
   reg_bhv reg7 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[7]), .clk(clk), .Q(data_out7));
   reg_bhv reg8 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[8]), .clk(clk), .Q(data_out8));
   reg_bhv reg9 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[9]), .clk(clk), .Q(data_out9));
   reg_bhv reg10 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[10]), .clk(clk), .Q(data_out10));
   reg_bhv reg11 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[11]), .clk(clk), .Q(data_out11));
   reg_bhv reg12 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[12]), .clk(clk), .Q(data_out12));
   reg_bhv reg13 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[13]), .clk(clk), .Q(data_out13));
   reg_bhv reg14 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[14]), .clk(clk), .Q(data_out14));
   reg_bhv reg15 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[15]), .clk(clk), .Q(data_out15));
   reg_bhv reg16 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[16]), .clk(clk), .Q(data_out16));
   reg_bhv reg17 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[17]), .clk(clk), .Q(data_out17));
   reg_bhv reg18 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[18]), .clk(clk), .Q(data_out18));
   reg_bhv reg19 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[19]), .clk(clk), .Q(data_out19));
   reg_bhv reg20 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[20]), .clk(clk), .Q(data_out20));
   reg_bhv reg21 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[21]), .clk(clk), .Q(data_out21));
   reg_bhv reg22 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[22]), .clk(clk), .Q(data_out22));
   reg_bhv reg23 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[23]), .clk(clk), .Q(data_out23));
   reg_bhv reg24 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[24]), .clk(clk), .Q(data_out24));
   reg_bhv reg25 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[25]), .clk(clk), .Q(data_out25));
   reg_bhv reg26 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[26]), .clk(clk), .Q(data_out26));
   reg_bhv reg27 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[27]), .clk(clk), .Q(data_out27));
   reg_bhv reg28 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[28]), .clk(clk), .Q(data_out28));
   reg_bhv reg29 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[29]), .clk(clk), .Q(data_out29));
   reg_bhv reg30 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[30]), .clk(clk), .Q(data_out30));
   reg_bhv reg31 (.D(wr_data), .wr_en(wr_en), .wr_reg(wr_reg[31]), .clk(clk), .Q(data_out31));

endmodule
```

```
module reg_array_multiplexer(read_addr, read_data, clk, data_in0, data_in1,
   data_in2, data_in3, data_in4, data_in5, data_in6, data_in7, data_in8, data_in9,
   data_in10, data_in11, data_in12, data_in13, data_in14, data_in15, data_in16,
   data_in17, data_in18, data_in19, data_in20, data_in21, data_in22, data_in23,
   data_in24, data_in25, data_in26, data_in27, data_in28, data_in29, data_in30,
   data_in31);

   input clk;
   input [4:0] read_addr; // Register address to read
   input [31:0] data_in0, data_in1, data_in2, data_in3, data_in4, data_in5,
   data_in6, data_in7, data_in8, data_in9, data_in10, data_in11, data_in12,
   data_in13, data_in14, data_in15, data_in16, data_in17, data_in18, data_in19,
   data_in20, data_in21, data_in22, data_in23, data_in24, data_in25, data_in26,
   data_in27, data_in28, data_in29, data_in30, data_in31;
   output reg [31:0] read_data;

   initial
      read_data = 0;

   // The read multiplexer receives data from each of the 32 registers in the register
   // file.  However, only one register's data is sent from the ID stage as output on
   // the lower-half of the clock cycle.
   always @(negedge clk)
      begin
         if(read_addr == 0)
            read_data = data_in0;
         else if(read_addr == 1)
            read_data = data_in1;
         else if(read_addr == 2)
            read_data = data_in2;
         else if(read_addr == 3)
            read_data = data_in3;
         else if(read_addr == 4)
            read_data = data_in4;
         else if(read_addr == 5)
            read_data = data_in5;
         else if(read_addr == 6)
            read_data = data_in6;
         else if(read_addr == 7)
            read_data = data_in7;
         else if(read_addr == 8)
            read_data = data_in8;
         else if(read_addr == 9)
            read_data = data_in9;
         else if(read_addr == 10)
            read_data = data_in10;
         else if(read_addr == 11)
            read_data = data_in11;
         else if(read_addr == 12)
            read_data = data_in12;
         else if(read_addr == 13)
            read_data = data_in13;
         else if(read_addr == 14)
            read_data = data_in14;
         else if(read_addr == 15)
            read_data = data_in15;
         else if(read_addr == 16)
            read_data = data_in16;
         else if(read_addr == 17)
            read_data = data_in17;
         else if(read_addr == 18)
            read_data = data_in18;
         else if(read_addr == 19)
            read_data = data_in19;
         else if(read_addr == 20)
            read_data = data_in20;
         else if(read_addr == 21)
            read_data = data_in21;
         else if(read_addr == 22)
            read_data = data_in22;
         else if(read_addr == 23)
            read_data = data_in23;
```
(continued on next page)

```
        else if(read_addr == 24)
           read_data = data_in24;
        else if(read_addr == 25)
           read_data = data_in25;
        else if(read_addr == 26)
           read_data = data_in26;
        else if(read_addr == 27)
           read_data = data_in27;
        else if(read_addr == 28)
           read_data = data_in28;
        else if(read_addr == 29)
           read_data = data_in29;
        else if(read_addr == 30)
           read_data = data_in30;
        else
           read_data = data_in31;
     end

endmodule
```

3. Sign Extension Module

```
module sign_extend_16to32bit(half_word, full_word);

   input [15:0] half_word;
   output reg [31:0] full_word;

   initial
      full_word = 0;

   // 16 zero bits are concatenated with the lower 16 bits of the instruction
   always @ (half_word)
      full_word = {16'b00000000_00000000, half_word};


endmodule
```

F. ID/EX REGISTER

```
module ID_EX_reg_tmr2(clk, regdst_a, regwrite_a, alusrc_a, memread_a, memwrite_a,
   memtoreg_a, beq_a, bne_a, blz_a, bgz_a, blez_a, bgez_a, jump_a, irqp_a, aluop_a,
   reg1_a, reg2_a, pc4_addr_a, sign_ext_a, jump_addr_a, wb_reg_addr1_a,
   wb_reg_addr2_a, shamt_a, fregdst_a, fregwrite_a, falusrc_a, fmemread_a,
   fmemwrite_a, fmemtoreg_a, fbeq_a, fbne_a, fblz_a, fbgz_a, fblez_a, fbgez_a,
   fjump_a, firqp_a, faluop_a, freg1_a, freg2_a, fpc4_addr_a, fsign_ext_a,
   fjump_addr_a, fwb_reg_addr1_a, fwb_reg_addr2_a, fshamt_a,
   regdst_b, regwrite_b, alusrc_b, memread_b, memwrite_b,
   memtoreg_b, beq_b, bne_b, blz_b, bgz_b, blez_b, bgez_b, jump_b, irqp_b, aluop_b,
   reg1_b, reg2_b, pc4_addr_b, sign_ext_b, jump_addr_b, wb_reg_addr1_b,
   wb_reg_addr2_b, shamt_b, fregdst_b, fregwrite_b, falusrc_b, fmemread_b,
   fmemwrite_b, fmemtoreg_b, fbeq_b, fbne_b, fblz_b, fbgz_b, fblez_b, fbgez_b,
   fjump_b, firqp_b, faluop_b, freg1_b, freg2_b, fpc4_addr_b, fsign_ext_b,
   fjump_addr_b, fwb_reg_addr1_b, fwb_reg_addr2_b, fshamt_b,
   regdst_c, regwrite_c, alusrc_c, memread_c, memwrite_c,
   memtoreg_c, beq_c, bne_c, blz_c, bgz_c, blez_c, bgez_c, jump_c, irqp_c, aluop_c,
   reg1_c, reg2_c, pc4_addr_c, sign_ext_c, jump_addr_c, wb_reg_addr1_c,
   wb_reg_addr2_c, shamt_c, fregdst_c, fregwrite_c, falusrc_c, fmemread_c,
   fmemwrite_c, fmemtoreg_c, fbeq_c, fbne_c, fblz_c, fbgz_c, fblez_c, fbgez_c,
   fjump_c, firqp_c, faluop_c, freg1_c, freg2_c, fpc4_addr_c, fsign_ext_c,
   fjump_addr_c, fwb_reg_addr1_c, fwb_reg_addr2_c, fshamt_c);
```
(continued on next page)

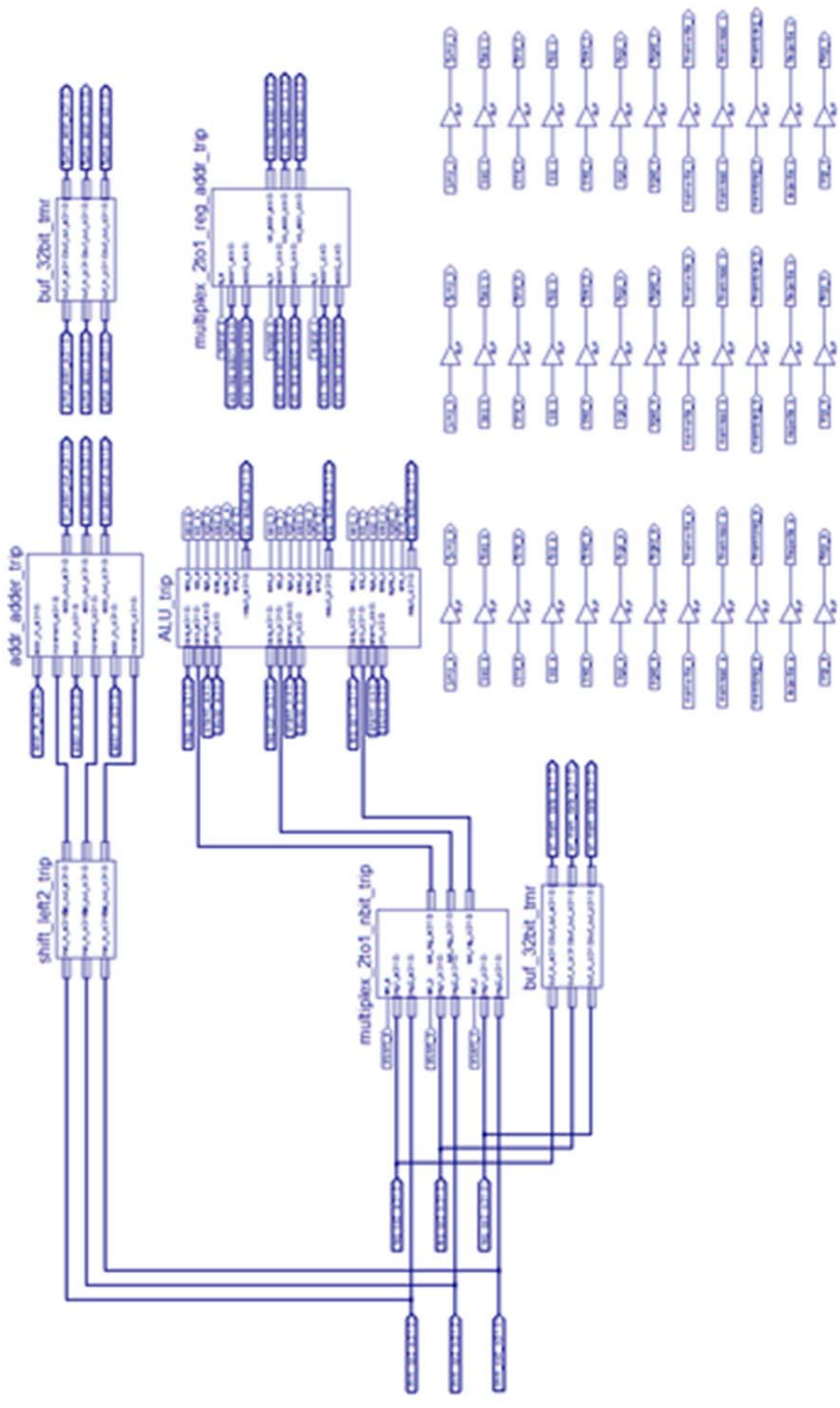
```
// All of the control flags decoded in the ID stage are forwarded to the EX stage
input clk;
input regdst_a, regdst_b, regdst_c;
input regwrite_a, regwrite_b, regwrite_c;
input alusrc_a, alusrc_b, alusrc_c;
input memread_a, memread_b, memread_c;
input memwrite_a, memwrite_b, memwrite_c;
input memtoreg_a, memtoreg_b, memtoreg_c;
input beq_a, beq_b, beq_c;
input bne_a, bne_b, bne_c;
input blz_a, blz_b, blz_c;
input bgz_a, bgz_b, bgz_c;
input blez_a, blez_b, blez_c;
input bgez_a, bgez_b, bgez_c;
input jump_a, jump_b, jump_c;
input irqp_a, irqp_b, irqp_c;
input [3:0] aluop_a, aluop_b, aluop_c;
input [4:0] wb_reg_addr1_a, wb_reg_addr1_b, wb_reg_addr1_c;
input [4:0] wb_reg_addr2_a, wb_reg_addr2_b, wb_reg_addr2_c;
input [4:0] shamt_a, shamt_b, shamt_c;
input [31:0] reg1_a, reg1_b, reg1_c;
input [31:0] reg2_a, reg2_b, reg2_c;
input [31:0] jump_addr_a, jump_addr_b, jump_addr_c;
input [31:0] pc4_addr_a, pc4_addr_b, pc4_addr_c;
input [31:0] sign_ext_a, sign_ext_b, sign_ext_c;

// Wires connecting pipeline registers to voters
wire nregdst_a, nregdst_b, nregdst_c;
wire nregwrite_a, nregwrite_b, nregwrite_c;
wire nalusrc_a, nalusrc_b, nalusrc_c;
wire nmemread_a, nmemread_b, nmemread_c;
wire nmemwrite_a, nmemwrite_b, nmemwrite_c;
wire nmemtoreg_a, nmemtoreg_b, nmemtoreg_c;
wire nbeq_a, nbeq_b, nbeq_c;
wire nbne_a, nbne_b, nbne_c;
wire nblz_a, nblz_b, nblz_c;
wire nbgz_a, nbgz_b, nbgz_c;
wire nblez_a, nblez_b, nblez_c;
wire nbgez_a, nbgez_b, nbgez_c;
wire njump_a, njump_b, njump_c;
wire nirqp_a, nirqp_b, nirqp_c;
wire [3:0] naluop_a, naluop_b, naluop_c;
wire [4:0] nwb_reg_addr1_a, nwb_reg_addr1_b, nwb_reg_addr1_c;
wire [4:0] nwb_reg_addr2_a, nwb_reg_addr2_b, nwb_reg_addr2_c;
wire [4:0] nshamt_a, nshamt_b, nshamt_c;
wire [31:0] nreg1_a, nreg1_b, nreg1_c;
wire [31:0] nreg2_a, nreg2_b, nreg2_c;
wire [31:0] njump_addr_a, njump_addr_b, njump_addr_c;
wire [31:0] npc4_addr_a, npc4_addr_b, npc4_addr_c;
wire [31:0] nsign_ext_a, nsign_ext_b, nsign_ext_c;
```

(continued on next page)

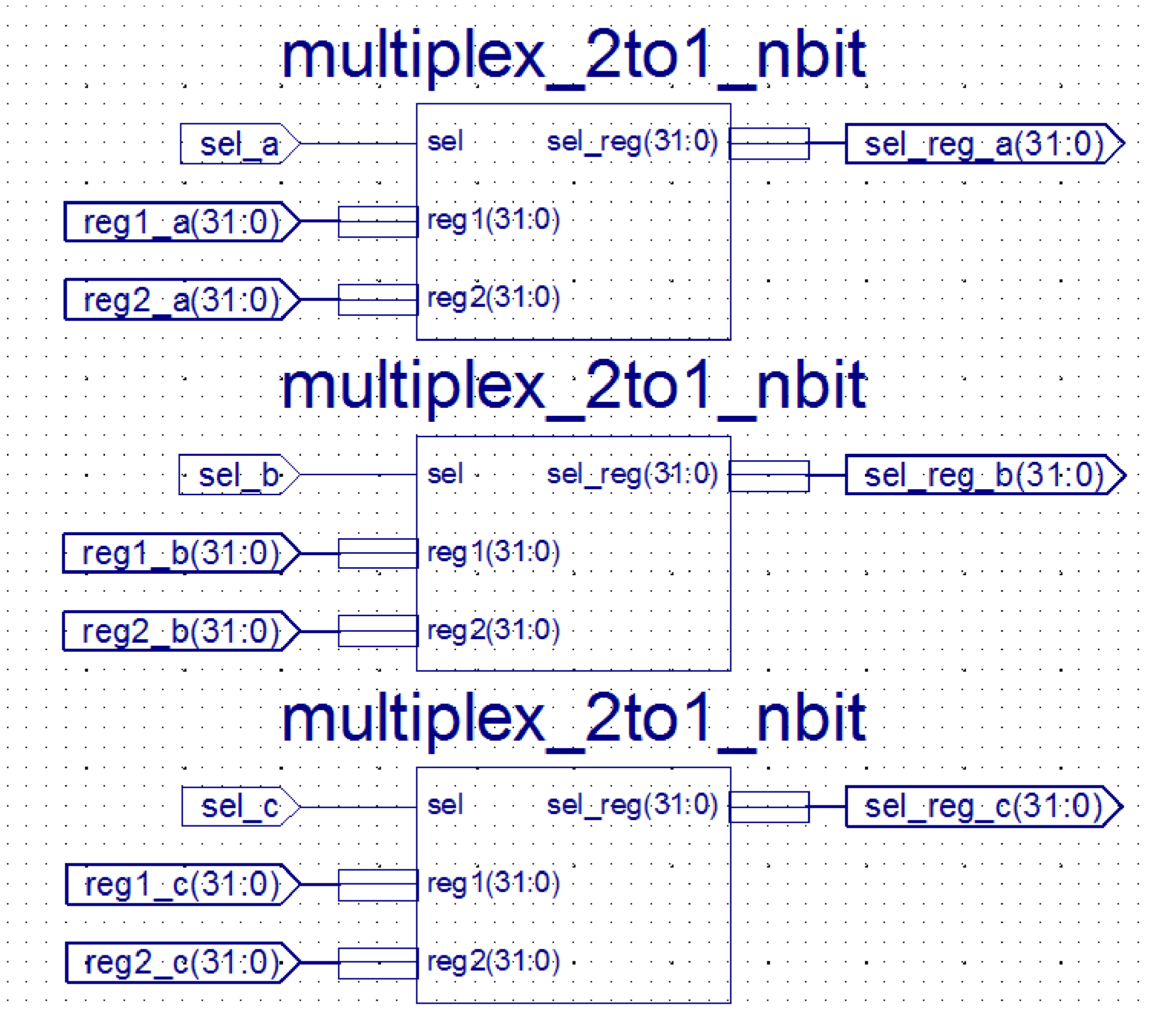
```
// Output of the voter circuits
output fregdst_a, fregdst_b, fregdst_c;
output fregwrite_a, fregwrite_b, fregwrite_c;
output falusrc_a, falusrc_b, falusrc_c;
output fmemread_a, fmemread_b, fmemread_c;
output fmemwrite_a, fmemwrite_b, fmemwrite_c;
output fmemtoreg_a, fmemtoreg_b, fmemtoreg_c;
output fbeq_a, fbeq_b, fbeq_c;
output fbne_a, fbne_b, fbne_c;
output fblz_a, fblz_b, fblz_c;
output fbgz_a, fbgz_b, fbgz_c;
output fblez_a, fblez_b, fblez_c;
output fbgez_a, fbgez_b, fbgez_c;
output fjump_a, fjump_b, fjump_c;
output firqp_a, firqp_b, firqp_c;
output [3:0] faluop_a, faluop_b, faluop_c;
output [4:0] fwb_reg_addr1_a, fwb_reg_addr1_b, fwb_reg_addr1_c;
output [4:0] fwb_reg_addr2_a, fwb_reg_addr2_b, fwb_reg_addr2_c;
output [4:0] fshamt_a, fshamt_b, fshamt_c;
output [31:0] freg1_a, freg1_b, freg1_c;
output [31:0] freg2_a, freg2_b, freg2_c;
output [31:0] fjump_addr_a, fjump_addr_b, fjump_addr_c;
output [31:0] fpc4_addr_a, fpc4_addr_b, fpc4_addr_c;
output [31:0] fsign_ext_a, fsign_ext_b, fsign_ext_c;

// Voter declaration
tmr_voter_trip voter0 (.a(nregdst_a), .b(nregdst_b), .c(nregdst_c),
.data_out_a(fregdst_a), .data_out_b(fregdst_b), .data_out_c(fregdst_c));
tmr_voter_trip voter1 (.a(nregwrite_a), .b(nregwrite_b), .c(nregwrite_c),
.data_out_a(fregwrite_a), .data_out_b(fregwrite_b), .data_out_c(fregwrite_c));
tmr_voter_trip voter2 (.a(nalusrc_a), .b(nalusrc_b), .c(nalusrc_c),
.data_out_a(falusrc_a), .data_out_b(falusrc_b), .data_out_c(falusrc_c));
tmr_voter_trip voter3 (.a(nmemread_a), .b(nmemread_b), .c(nmemread_c),
.data_out_a(fmemread_a), .data_out_b(fmemread_b), .data_out_c(fmemread_c));
tmr_voter_trip voter4 (.a(nmemwrite_a), .b(nmemwrite_b), .c(nmemwrite_c),
.data_out_a(fmemwrite_a), .data_out_b(fmemwrite_b), .data_out_c(fmemwrite_c));
tmr_voter_trip voter5 (.a(nmemtoreg_a), .b(nmemtoreg_b), .c(nmemtoreg_c),
.data_out_a(fmemtoreg_a), .data_out_b(fmemtoreg_b), .data_out_c(fmemtoreg_c));
tmr_voter_trip voter6 (.a(nbeq_a), .b(nbeq_b), .c(nbeq_c),
.data_out_a(fbeq_a), .data_out_b(fbeq_b), .data_out_c(fbeq_c));
tmr_voter_trip voter7 (.a(nbne_a), .b(nbne_b), .c(nbne_c),
.data_out_a(fbne_a), .data_out_b(fbne_b), .data_out_c(fbne_c));
tmr_voter_trip voter8 (.a(nblz_a), .b(nblz_b), .c(nblz_c),
.data_out_a(fblz_a), .data_out_b(fblz_b), .data_out_c(fblz_c));
tmr_voter_trip voter9 (.a(nbgz_a), .b(nbgz_b), .c(nbgz_c),
.data_out_a(fbgz_a), .data_out_b(fbgz_b), .data_out_c(fbgz_c));
tmr_voter_trip voter10 (.a(nblez_a), .b(nblez_b), .c(nblez_c),
.data_out_a(fblez_a), .data_out_b(fblez_b), .data_out_c(fblez_c));
tmr_voter_trip voter11 (.a(nbgez_a), .b(nbgez_b), .c(nbgez_c),
.data_out_a(fbgez_a), .data_out_b(fbgez_b), .data_out_c(fbgez_c));
tmr_voter_trip voter12 (.a(njump_a), .b(njump_b), .c(njump_c),
.data_out_a(fjump_a), .data_out_b(fjump_b), .data_out_c(fjump_c));
tmr_voter_trip voter13 (.a(nirqp_a), .b(nirqp_b), .c(nirqp_c),
.data_out_a(firqp_a), .data_out_b(firqp_b), .data_out_c(firqp_c));
tmr_voter_4bit_trip voter14 (.a(naluop_a), .b(naluop_b), .c(naluop_c),
.data_out_a(faluop_a), .data_out_b(faluop_b), .data_out_c(faluop_c));
tmr_voter_5bit_trip voter15 (.a(nwb_reg_addr1_a), .b(nwb_reg_addr1_b), .c(nwb_reg_addr1_c),
.data_out_a(fwb_reg_addr1_a), .data_out_b(fwb_reg_addr1_b), .data_out_c(fwb_reg_addr1_c));
tmr_voter_5bit_trip voter16 (.a(nwb_reg_addr2_a), .b(nwb_reg_addr2_b), .c(nwb_reg_addr2_c),
.data_out_a(fwb_reg_addr2_a), .data_out_b(fwb_reg_addr2_b), .data_out_c(fwb_reg_addr2_c));
tmr_voter_5bit_trip voter17 (.a(nshamt_a), .b(nshamt_b), .c(nshamt_c),
.data_out_a(fshamt_a), .data_out_b(fshamt_b), .data_out_c(fshamt_c));
tmr_voter_32bit_trip voter18 (.a(nreg1_a), .b(nreg1_b), .c(nreg1_c),
.data_out_a(freg1_a), .data_out_b(freg1_b), .data_out_c(freg1_c));
tmr_voter_32bit_trip voter19 (.a(nreg2_a), .b(nreg2_b), .c(nreg2_c),
.data_out_a(freg2_a), .data_out_b(freg2_b), .data_out_c(freg2_c));
tmr_voter_32bit_trip voter20 (.a(njump_addr_a), .b(njump_addr_b), .c(njump_addr_c),
.data_out_a(fjump_addr_a), .data_out_b(fjump_addr_b), .data_out_c(fjump_addr_c));
tmr_voter_32bit_trip voter21 (.a(npc4_addr_a), .b(npc4_addr_b), .c(npc4_addr_c),
.data_out_a(fpc4_addr_a), .data_out_b(fpc4_addr_b), .data_out_c(fpc4_addr_c));
tmr_voter_32bit_trip voter22 (.a(nsign_ext_a), .b(nsign_ext_b), .c(nsign_ext_c),
.data_out_a(fsign_ext_a), .data_out_b(fsign_ext_b), .data_out_c(fsign_ext_c));
```

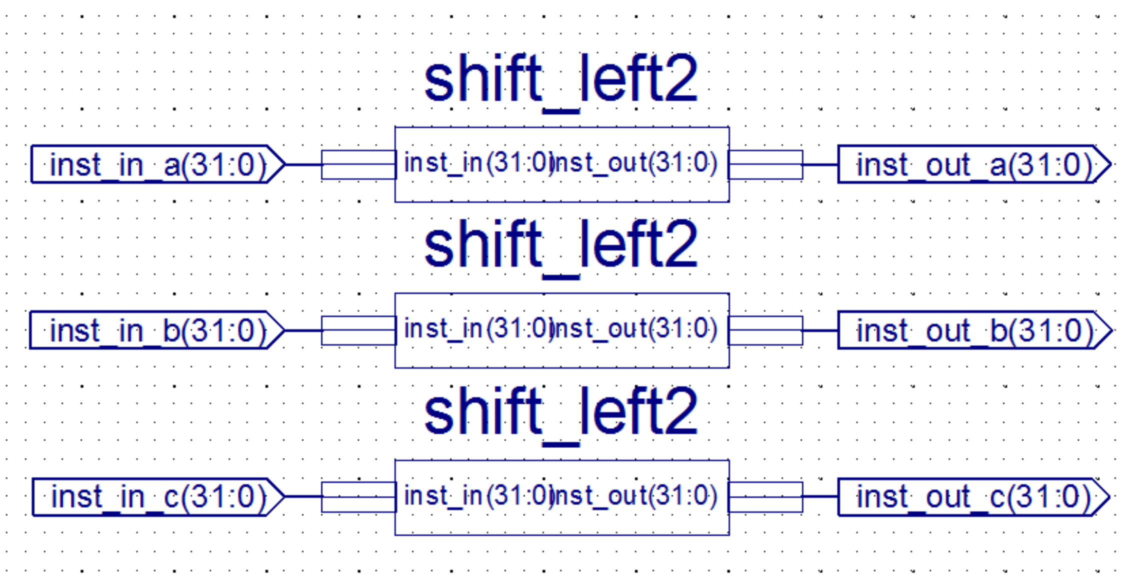
```verilog
    // Pipeline register declaration
    ID_EX_reg reg_a (.clk(clk), .regdst(regdst_a), .regwrite(regwrite_a),
    .alusrc(alusrc_a), .memread(memread_a), .memwrite(memwrite_a), .memtoreg(memtoreg_a),
    .aluop(aluop_a), .bne(bne_a), .beq(beq_a), .blz(blz_a), .bgz(bgz_a), .blez(blez_a),
    .bgez(bgez_a), .jump(jump_a), .irqp(irqp_a), .reg1(reg1_a), .reg2(reg2_a),
    .pc4_addr(pc4_addr_a), .sign_ext(sign_ext_a), .jump_addr(jump_addr_a),
    .wb_reg_addr1(wb_reg_addr1_a), .wb_reg_addr2(wb_reg_addr2_a), .shamt(shamt_a),
    .fregdst(nregdst_a), .fregwrite(nregwrite_a), .falusrc(nalusrc_a),
    .fmemread(nmemread_a), .fmemwrite(nmemwrite_a), .fmemtoreg(nmemtoreg_a),
    .faluop(naluop_a), .fbne(nbne_a), .fbeq(nbeq_a), .fblz(nblz_a), .fbgz(nbgz_a),
    .fblez(nblez_a), .fbgez(nbgez_a), .fjump(njump_a), .firqp(nirqp_a), .freg1(nreg1_a),
    .freg2(nreg2_a), .fpc4_addr(npc4_addr_a), .fsign_ext(nsign_ext_a),
    .fjump_addr(njump_addr_a), .fwb_reg_addr1(nwb_reg_addr1_a),
    .fwb_reg_addr2(nwb_reg_addr2_a), .fshamt(nshamt_a));

    ID_EX_reg reg_b (.clk(clk), .regdst(regdst_b), .regwrite(regwrite_b),
    .alusrc(alusrc_b), .memread(memread_b), .memwrite(memwrite_b), .memtoreg(memtoreg_b),
    .aluop(aluop_b), .bne(bne_b), .beq(beq_b), .blz(blz_b), .bgz(bgz_b), .blez(blez_b),
    .bgez(bgez_b), .jump(jump_b), .irqp(irqp_b), .reg1(reg1_b), .reg2(reg2_b),
    .pc4_addr(pc4_addr_b), .sign_ext(sign_ext_b), .jump_addr(jump_addr_b),
    .wb_reg_addr1(wb_reg_addr1_b), .wb_reg_addr2(wb_reg_addr2_b), .shamt(shamt_b),
    .fregdst(nregdst_b), .fregwrite(nregwrite_b), .falusrc(nalusrc_b),
    .fmemread(nmemread_b), .fmemwrite(nmemwrite_b), .fmemtoreg(nmemtoreg_b),
    .faluop(naluop_b), .fbne(nbne_b), .fbeq(nbeq_b),   .fblz(nblz_b), .fbgz(nbgz_b),
    .fblez(nblez_b), .fbgez(nbgez_b), .fjump(njump_b), .firqp(nirqp_b), .freg1(nreg1_b),
    .freg2(nreg2_b), .fpc4_addr(npc4_addr_b), .fsign_ext(nsign_ext_b),
    .fjump_addr(njump_addr_b), .fwb_reg_addr1(nwb_reg_addr1_b),
    .fwb_reg_addr2(nwb_reg_addr2_b), .fshamt(nshamt_b));

    ID_EX_reg reg_c (.clk(clk), .regdst(regdst_c), .regwrite(regwrite_c),
    .alusrc(alusrc_c), .memread(memread_c), .memwrite(memwrite_c), .memtoreg(memtoreg_c),
    .aluop(aluop_c), .bne(bne_c), .beq(beq_c), .blz(blz_c), .bgz(bgz_c), .blez(blez_c),
    .bgez(bgez_c), .jump(jump_c), .irqp(irqp_c), .reg1(reg1_c), .reg2(reg2_c),
    .pc4_addr(pc4_addr_c), .sign_ext(sign_ext_c), .jump_addr(jump_addr_c),
    .wb_reg_addr1(wb_reg_addr1_c), .wb_reg_addr2(wb_reg_addr2_c), .shamt(shamt_c),
    .fregdst(nregdst_c), .fregwrite(nregwrite_c), .falusrc(nalusrc_c),
    .fmemread(nmemread_c), .fmemwrite(nmemwrite_c), .fmemtoreg(nmemtoreg_c),
    .faluop(naluop_c), .fbne(nbne_c), .fbeq(nbeq_c),   .fblz(nblz_c), .fbgz(nbgz_c),
    .fblez(nblez_c), .fbgez(nbgez_c), .fjump(njump_c), .firqp(nirqp_c), .freg1(nreg1_c),
    .freg2(nreg2_c), .fpc4_addr(npc4_addr_c), .fsign_ext(nsign_ext_c),
    .fjump_addr(njump_addr_c), .fwb_reg_addr1(nwb_reg_addr1_c),
    .fwb_reg_addr2(nwb_reg_addr2_c), .fshamt(nshamt_c));

endmodule
```

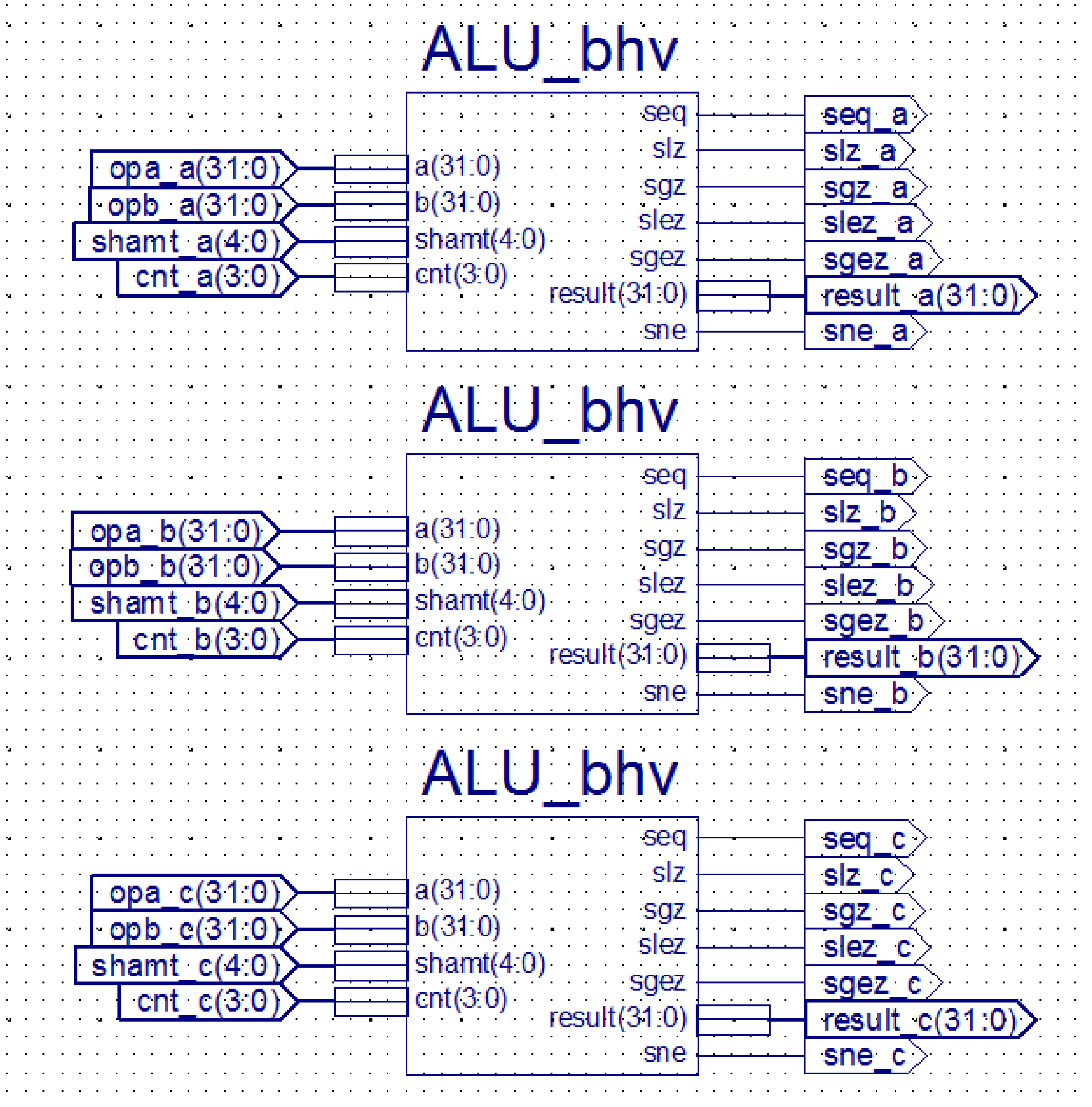
G. EX STAGE

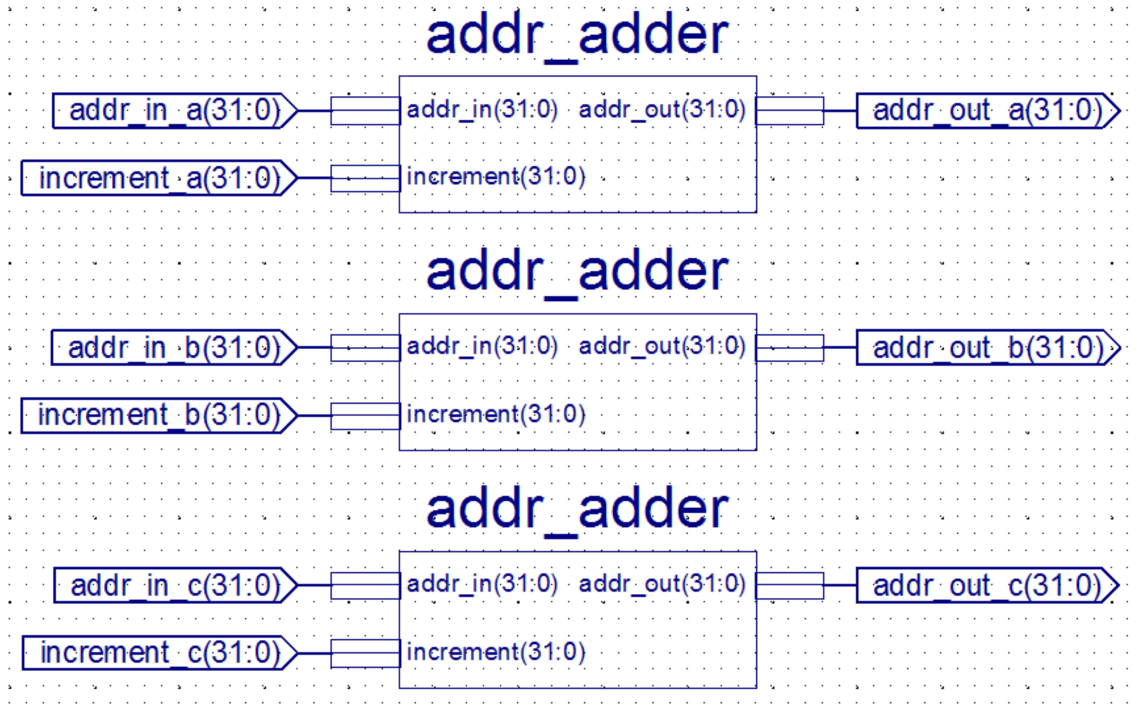


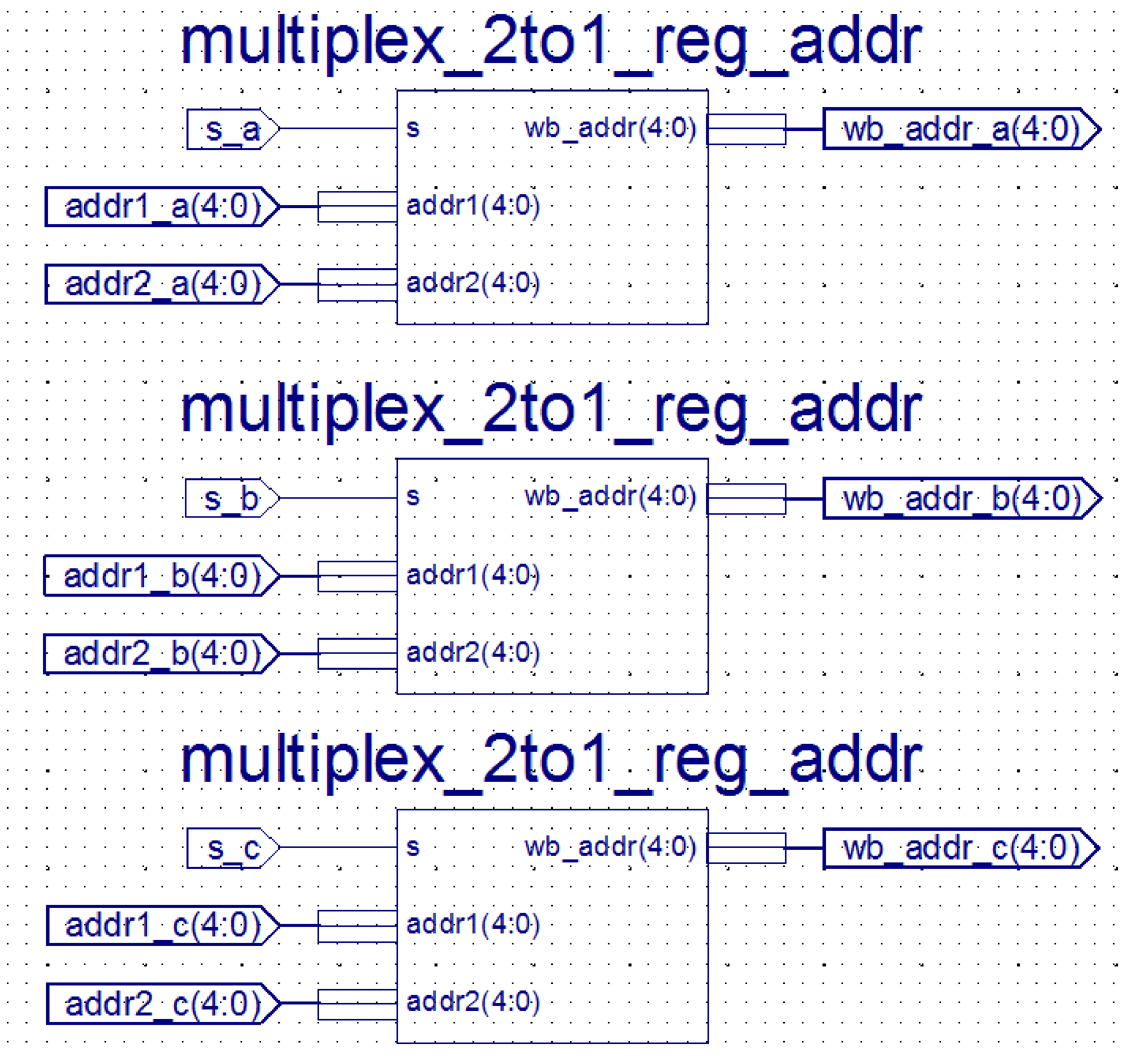
buf_32bit_tmr
multiplex_2to1_reg_addr_trip
addr_adder_trip
ALU_trip
shift_left2_trip
multiplex_2to1_nbit_trip
buf_32bit_tmr


multiplex_2to1_nbit
sel_a
sel
sel_reg(31:0)
sel_reg_a(31:0)
reg1_a(31:0)
reg1(31:0)
reg2_a(31:0)
reg2(31:0)
multiplex_2to1_nbit
sel_b
sel
sel_reg(31:0)
sel_reg_b(31:0)
reg1_b(31:0)
reg1(31:0)
reg2_b(31:0)
reg2(31:0)
multiplex_2to1_nbit
sel_c
sel
sel_reg(31:0)
sel_reg_c(31:0)
reg1_c(31:0)
reg1(31:0)
reg2_c(31:0)
reg2(31:0)


shift_left2
inst_in_a(31:0)
inst_in(31:0)
inst_out(31:0)
inst_out_a(31:0)
shift_left2
inst_in_b(31:0)
inst_in(31:0)
inst_out(31:0)
inst_out_b(31:0)
shift_left2
inst_in_c(31:0)
inst_in(31:0)
inst_out(31:0)
inst_out_c(31:0)


ALU_bhv
opa_a(31:0)
opb_a(31:0)
shamt_a(4:0)
cnt_a(3:0)
a(31:0)
b(31:0)
shamt(4:0)
cnt(3:0)
seq
slz
sgz
slez
sgez
result(31:0)
sne
seq_a
slz_a
sgz_a
slez_a
sgez_a
result_a(31:0)
sne_a
ALU_bhv
opa_b(31:0)
opb_b(31:0)
shamt_b(4:0)
cnt_b(3:0)
a(31:0)
b(31:0)
shamt(4:0)
cnt(3:0)
seq
slz
sgz
slez
sgez
result(31:0)
sne
seq_b
slz_b
sgz_b
slez_b
sgez_b
result_b(31:0)
sne_b
ALU_bhv
opa_c(31:0)
opb_c(31:0)
shamt_c(4:0)
cnt_c(3:0)
a(31:0)
b(31:0)
shamt(4:0)
cnt(3:0)
seq
slz
sgz
slez
sgez
result(31:0)
sne
seq_c
slz_c
sgz_c
slez_c
sgez_c
result_c(31:0)
sne_c


addr_adder
addr_in_a(31:0)
increment_a(31:0)
addr_in(31:0)
increment(31:0)
addr_out(31:0)
addr_out_a(31:0)
addr_adder
addr_in_b(31:0)
increment_b(31:0)
addr_in(31:0)
increment(31:0)
addr_out(31:0)
addr_out_b(31:0)
addr_adder
addr_in_c(31:0)
increment_c(31:0)
addr_in(31:0)
increment(31:0)
addr_out(31:0)
addr_out_c(31:0)


multiplex_2to1_reg_addr
s_a
s
wb_addr(4:0)
wb_addr_a(4:0)
addr1_a(4:0)
addr1(4:0)
addr2_a(4:0)
addr2(4:0)
multiplex_2to1_reg_addr
s_b
s
wb_addr(4:0)
wb_addr_b(4:0)
addr1_b(4:0)
addr1(4:0)
addr2_b(4:0)
addr2(4:0)
multiplex_2to1_reg_addr
s_c
s
wb_addr(4:0)
wb_addr_c(4:0)
addr1_c(4:0)
addr1(4:0)
addr2_c(4:0)
addr2(4:0)

1. ALU (ALU_bhv)

```
module ALU_bhv(a, b, shamt, cnt, result, seq, sne, slz, slez, sgz, sgez);

   input [31:0] a, b; // ALU operands
   input [4:0] shamt; // Number of spaces to shift in SLV and SRV instruction
   input [3:0] cnt; // Indicates the type of ALU operation to be performed
   output reg [31:0] result; // Output result
   output reg seq, sne, slz, slez, sgz, sgez; // Comparison bits

   initial
      begin
         result <= 0;
         seq <= 0;
         sne <= 0;
         slz <= 0;
         slez <= 0;
         sgz <= 0;
         sgez <= 0;
      end

   // Set ALU comparison bits seq and sne
   always @ (a or b or cnt)
      begin
         if (a == b)
            begin
               seq = 1'b1;
               sne = 1'b0;
            end
         else
            begin
               seq = 1'b0;
               sne = 1'b1;
            end
      end

   // Set ALU comparison bits sgz, sgez, slz, and slez
   always @ (a or b or cnt)
      begin
         if (a[31] == 0)
            begin
               if (a == 0)
                  begin
                     sgz = 0;
                     sgez = 1;
                     slz = 0;
                     slez = 1;
                  end
               else
                  begin
                     sgz = 1;
                     sgez = 1;
                     slz = 0;
                     slez = 0;
                  end
            end
         if (a[31] == 1)
            begin
               sgz = 0;
               sgez = 0;
               slz = 1;
               slez = 1;
            end
      end
```

(continued on next page)

```verilog
   // ALU Operations
   always @(a or b or cnt)
      begin
         if (cnt == 0)
            result = a + b; // Addition

         else if (cnt == 1)
            result = a - b; // Subtraction

         else if (cnt == 2)
            result = a & b; // AND

         else if (cnt == 3)
            result = a | b; // OR

         else if (cnt == 4)
            result = a ^ b; // XOR

         else if (cnt == 5)
            result = ~(a | b); // NOR

         else if (cnt == 6)
            result = a << b; // SLLV

         else if (cnt == 7)
            result = a << shamt; // SLL

         else if (cnt == 8)
            result = a >> b; // SRLV

         else if (cnt == 9)
            result = a >> shamt; // SRL

         else if (cnt == 10) // SLT
            begin
               if (a < b)
                  result = 1;
               else
                  result = 0;
            end

      end

endmodule
```

2. Address Adder (addr_adder)

```
module addr_adder(addr_in, increment, addr_out);

   input [31:0] addr_in, increment;
   output reg [31:0] addr_out;

   initial
      addr_out = 0;

   // The offset is added to the PC + 4 address to obtain the potential branch
   // address result.  This result will only be forwarded if the criterion for a
   // branch instruction are met in the MEM stage.
   always @ (addr_in or increment)
      addr_out = addr_in + increment;


endmodule
```

3. Second ALU Operand Multiplexer (multiplex_2to1_nbit)

```
module multiplex_2to1_nbit(reg1, reg2, sel, sel_reg);

   input [31:0] reg1, reg2;
   input sel;
   output reg [31:0] sel_reg;

   initial
      sel_reg = 0;

   // Select either the second register file output value or the sign extended
   // immediate value to use as an operand to the ALU.
   always @(reg1 or reg2 or sel)
      begin
         if (sel == 1'b0)
            sel_reg = reg1;
         else
            sel_reg = reg2;
      end


endmodule
```

4. Immediate Offset Shifting Module (shift_left2)

```
module shift_left2(inst_in, inst_out);

   input [31:0] inst_in;
   output reg [31:0] inst_out;

   initial
      inst_out = 0;

   // Shifts the immediate field of the instruction left two bits to translate it
   // from a word offset to a byte offset.
   always @ (inst_in)
      inst_out = inst_in << 2;


endmodule
```

5. Register Address Multiplexer (multiplex_2to1_reg_addr)

```
module multiplex_2to1_reg_addr(addr1, addr2, s, wb_addr);

   input [4:0] addr1;
   input [4:0] addr2;
   input s;
   output reg [4:0] wb_addr;

   initial
      wb_addr = 0;

   always @ (addr1 or addr2 or s)
      begin
         if (s == 1'b0)
            wb_addr = addr1;
         else
            wb_addr = addr2;
      end


endmodule
```

H. EX/MEM REGISTER

```
module EX_MEM_reg_tmr2(memread_a, memwrite_a, memtoreg_a, regwrite_a, beq_a, bne_a,
   blz_a, bgz_a, blez_a, bgez_a, jump_a, seq_a, sne_a, slz_a, sgz_a, slez_a, sgez_a,
   irqp_a, br_addr_a, jump_addr_a, alu_result_a, wr_data_a, wb_reg_addr_a,
   fmemread_a, fmemwrite_a, fmemtoreg_a, fregwrite_a, fbeq_a, fbne_a,
   fblz_a, fbgz_a, fblez_a, fbgez_a, fjump_a, fseq_a, fsne_a, fslz_a, fsgz_a,
   fslez_a, fsgez_a, firqp_a, fbr_addr_a, fjump_addr_a, falu_result_a, fwr_data_a,
   fwb_reg_addr_a,
   memread_b, memwrite_b, memtoreg_b, regwrite_b, beq_b, bne_b, blz_b, bgz_b,
   blez_b, bgez_b, jump_b, seq_b, sne_b, slz_b, sgz_b, slez_b, sgez_b, irqp_b, br_addr_b,
   jump_addr_b, alu_result_b, wr_data_b, wb_reg_addr_b, fmemread_b, fmemwrite_b,
   fmemtoreg_b, fregwrite_b, fbeq_b, fbne_b, fblz_b, fbgz_b, fblez_b, fbgez_b,
   fjump_b, fseq_b, fsne_b, fslz_b, fsgz_b, fslez_b, fsgez_b,  firqp_b, fbr_addr_b,
   fjump_addr_b, falu_result_b, fwr_data_b, fwb_reg_addr_b,
   memread_c, memwrite_c, memtoreg_c, regwrite_c, beq_c, bne_c, blz_c, bgz_c,
   blez_c, bgez_c, jump_c, seq_c, sne_c, slz_c, sgz_c, slez_c, sgez_c, irqp_c, br_addr_c,
   jump_addr_c, alu_result_c, wr_data_c, wb_reg_addr_c, fmemread_c, fmemwrite_c,
   fmemtoreg_c, fregwrite_c, fbeq_c, fbne_c, fblz_c, fbgz_c, fblez_c, fbgez_c,
   fjump_c, fseq_c, fsne_c, fslz_c, fsgz_c, fslez_c, fsgez_c,  firqp_c, fbr_addr_c,
   fjump_addr_c, falu_result_c, fwr_data_c, fwb_reg_addr_c, clk);
```

(continued on next page)

```
// Inputs
input clk;
input memread_a, memread_b, memread_c;
input memwrite_a, memwrite_b, memwrite_c;
input memtoreg_a, memtoreg_b, memtoreg_c;
input regwrite_a, regwrite_b, regwrite_c;
input beq_a, beq_b, beq_c;
input bne_a, bne_b, bne_c;
input blz_a, blz_b, blz_c;
input bgz_a, bgz_b, bgz_c;
input blez_a, blez_b, blez_c;
input bgez_a, bgez_b, bgez_c;
input jump_a, jump_b, jump_c;
input seq_a, seq_b, seq_c;
input sne_a, sne_b, sne_c;
input slz_a, slz_b, slz_c;
input sgz_a, sgz_b, sgz_c;
input slez_a, slez_b, slez_c;
input sgez_a, sgez_b, sgez_c;
input irqp_a, irqp_b, irqp_c;
input [31:0] br_addr_a, br_addr_b, br_addr_c;
input [31:0] jump_addr_a, jump_addr_b, jump_addr_c;
input [31:0] alu_result_a, alu_result_b, alu_result_c;
input [31:0] wr_data_a, wr_data_b, wr_data_c;
input [4:0] wb_reg_addr_a, wb_reg_addr_b, wb_reg_addr_c;

// Wires connecting pipeline registers to voters
wire nmemread_a, nmemread_b, nmemread_c;
wire nmemwrite_a, nmemwrite_b, nmemwrite_c;
wire nmemtoreg_a, nmemtoreg_b, nmemtoreg_c;
wire nregwrite_a, nregwrite_b, nregwrite_c;
wire nbeq_a, nbeq_b, nbeq_c;
wire nbne_a, nbne_b, nbne_c;
wire nblz_a, nblz_b, nblz_c;
wire nbgz_a, nbgz_b, nbgz_c;
wire nblez_a, nblez_b, nblez_c;
wire nbgez_a, nbgez_b, nbgez_c;
wire njump_a, njump_b, njump_c;
wire nseq_a, nseq_b, nseq_c;
wire nsne_a, nsne_b, nsne_c;
wire nslz_a, nslz_b, nslz_c;
wire nsgz_a, nsgz_b, nsgz_c;
wire nslez_a, nslez_b, nslez_c;
wire nsgez_a, nsgez_b, nsgez_c;
wire nirqp_a, nirqp_b, nirqp_c;
wire [31:0] nbr_addr_a, nbr_addr_b, nbr_addr_c;
wire [31:0] njump_addr_a, njump_addr_b, njump_addr_c;
wire [31:0] nalu_result_a, nalu_result_b, nalu_result_c;
wire [31:0] nwr_data_a, nwr_data_b, nwr_data_c;
wire [4:0] nwb_reg_addr_a, nwb_reg_addr_b, nwb_reg_addr_c;
```

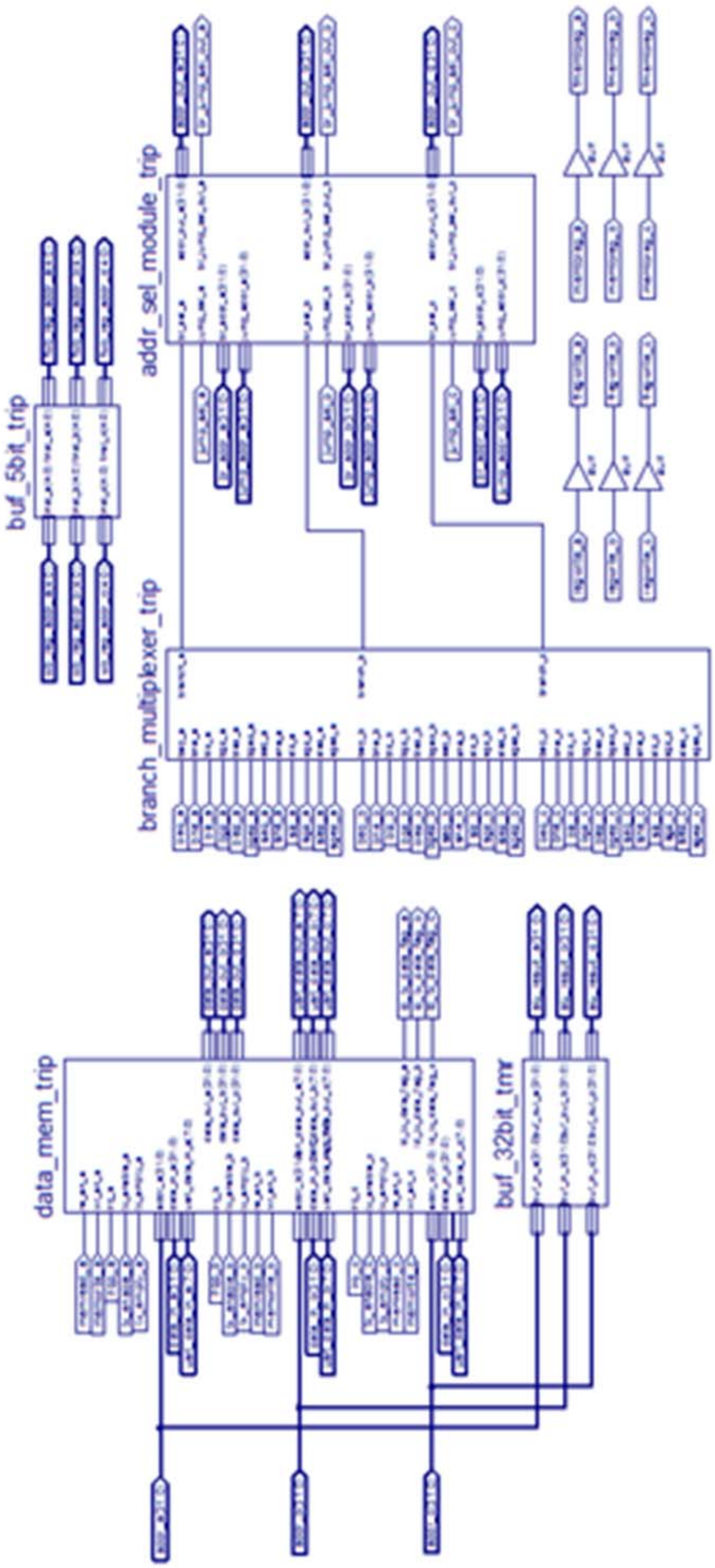
(continued on next page)

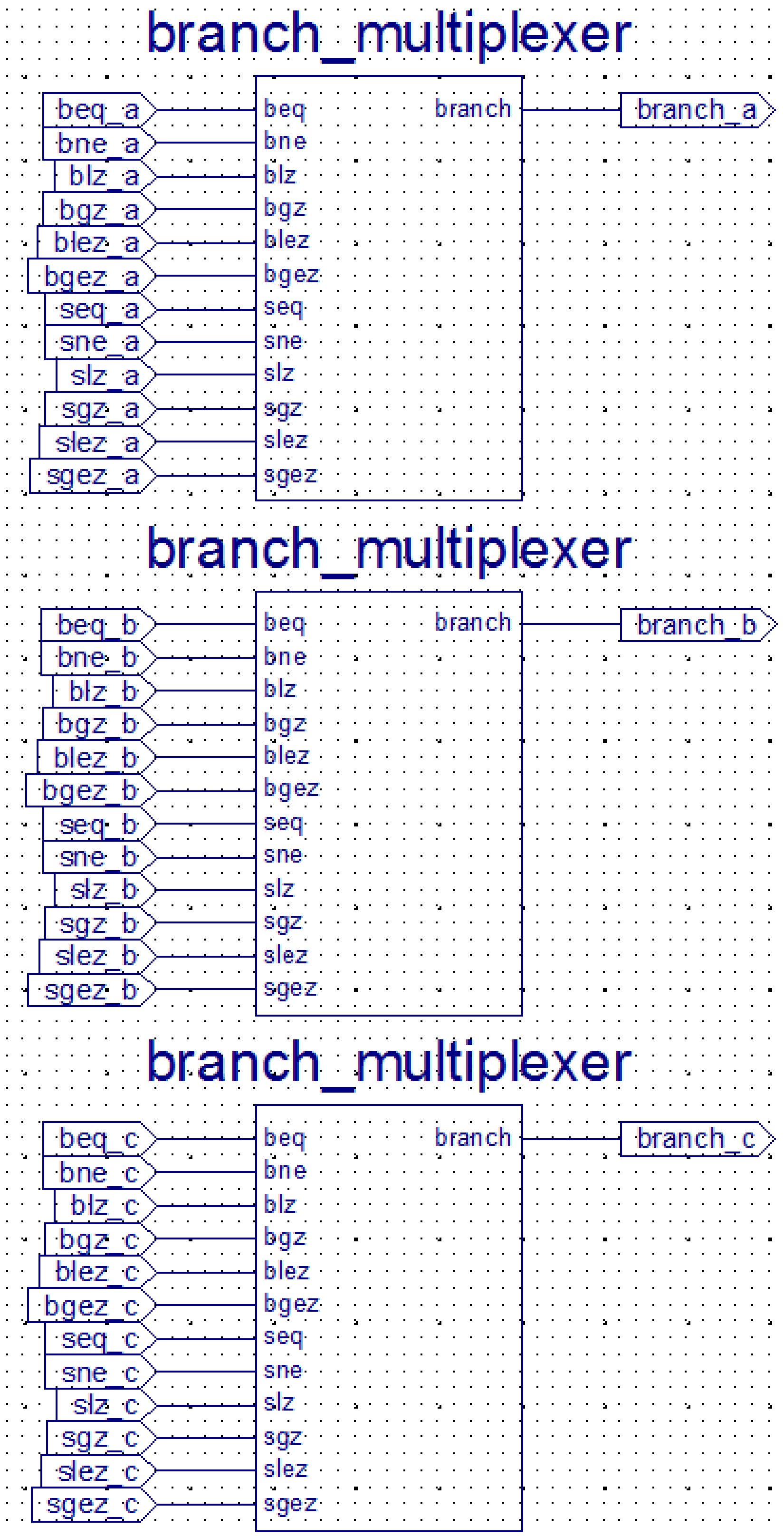
```
// Outputs
output fmemread_a, fmemread_b, fmemread_c;
output fmemwrite_a, fmemwrite_b, fmemwrite_c;
output fmemtoreg_a, fmemtoreg_b, fmemtoreg_c;
output fregwrite_a, fregwrite_b, fregwrite_c;
output fbeq_a, fbeq_b, fbeq_c;
output fbne_a, fbne_b, fbne_c;
output fblz_a, fblz_b, fblz_c;
output fbgz_a, fbgz_b, fbgz_c;
output fblez_a, fblez_b, fblez_c;
output fbgez_a, fbgez_b, fbgez_c;
output fjump_a, fjump_b, fjump_c;
output fseq_a, fseq_b, fseq_c;
output fsne_a, fsne_b, fsne_c;
output fslz_a, fslz_b, fslz_c;
output fsgz_a, fsgz_b, fsgz_c;
output fslez_a, fslez_b, fslez_c;
output fsgez_a, fsgez_b, fsgez_c;
output firqp_a, firqp_b, firqp_c;
output [31:0] fbr_addr_a, fbr_addr_b, fbr_addr_c;
output [31:0] fjump_addr_a, fjump_addr_b, fjump_addr_c;
output [31:0] falu_result_a, falu_result_b, falu_result_c;
output [31:0] fwr_data_a, fwr_data_b, fwr_data_c;
output [4:0] fwb_reg_addr_a, fwb_reg_addr_b, fwb_reg_addr_c;

// Voter declaration
tmr_voter_trip voter0 (.a(nmemread_a), .b(nmemread_b), .c(nmemread_c),
.data_out_a(fmemread_a), .data_out_b(fmemread_b), .data_out_c(fmemread_c));
tmr_voter_trip voter1 (.a(nmemwrite_a), .b(nmemwrite_b), .c(nmemwrite_c),
.data_out_a(fmemwrite_a), .data_out_b(fmemwrite_b), .data_out_c(fmemwrite_c));
tmr_voter_trip voter2 (.a(nmemtoreg_a), .b(nmemtoreg_b), .c(nmemtoreg_c),
.data_out_a(fmemtoreg_a), .data_out_b(fmemtoreg_b), .data_out_c(fmemtoreg_c));
tmr_voter_trip voter3 (.a(nregwrite_a), .b(nregwrite_b), .c(nregwrite_c),
.data_out_a(fregwrite_a), .data_out_b(fregwrite_b), .data_out_c(fregwrite_c));
tmr_voter_trip voter4 (.a(nbeq_a), .b(nbeq_b), .c(nbeq_c),
.data_out_a(fbeq_a), .data_out_b(fbeq_b), .data_out_c(fbeq_c));
tmr_voter_trip voter5 (.a(nbne_a), .b(nbne_b), .c(nbne_c),
.data_out_a(fbne_a), .data_out_b(fbne_b), .data_out_c(fbne_c));
tmr_voter_trip voter6 (.a(nblz_a), .b(nblz_b), .c(nblz_c),
.data_out_a(fblz_a), .data_out_b(fblz_b), .data_out_c(fblz_c));
tmr_voter_trip voter7 (.a(nbgz_a), .b(nbgz_b), .c(nbgz_c),
.data_out_a(fbgz_a), .data_out_b(fbgz_b), .data_out_c(fbgz_c));
tmr_voter_trip voter8 (.a(nblez_a), .b(nblez_b), .c(nblez_c),
.data_out_a(fblez_a), .data_out_b(fblez_b), .data_out_c(fblez_c));
tmr_voter_trip voter9 (.a(nbgez_a), .b(nbgez_b), .c(nbgez_c),
.data_out_a(fbgez_a), .data_out_b(fbgez_b), .data_out_c(fbgez_c));
tmr_voter_trip voter10 (.a(nseq_a), .b(nseq_b), .c(nseq_c),
.data_out_a(fseq_a), .data_out_b(fseq_b), .data_out_c(fseq_c));
tmr_voter_trip voter11 (.a(nsne_a), .b(nsne_b), .c(nsne_c),
.data_out_a(fsne_a), .data_out_b(fsne_b), .data_out_c(fsne_c));
tmr_voter_trip voter12 (.a(nslz_a), .b(nslz_b), .c(nslz_c),
.data_out_a(fslz_a), .data_out_b(fslz_b), .data_out_c(fslz_c));
tmr_voter_trip voter13 (.a(nsgz_a), .b(nsgz_b), .c(nsgz_c),
.data_out_a(fsgz_a), .data_out_b(fsgz_b), .data_out_c(fsgz_c));
tmr_voter_trip voter14 (.a(nslez_a), .b(nslez_b), .c(nslez_c),
.data_out_a(fslez_a), .data_out_b(fslez_b), .data_out_c(fslez_c));
tmr_voter_trip voter15 (.a(nsgez_a), .b(nsgez_b), .c(nsgez_c),
.data_out_a(fsgez_a), .data_out_b(fsgez_b), .data_out_c(fsgez_c));
tmr_voter_trip voter16 (.a(nirqp_a), .b(nirqp_b), .c(nirqp_c),
.data_out_a(firqp_a), .data_out_b(firqp_b), .data_out_c(firqp_c));
tmr_voter_trip voter17 (.a(njump_a), .b(njump_b), .c(njump_c),
.data_out_a(fjump_a), .data_out_b(fjump_b), .data_out_c(fjump_c));
tmr_voter_32bit_trip voter18 (.a(nbr_addr_a), .b(nbr_addr_b), .c(nbr_addr_c),
.data_out_a(fbr_addr_a), .data_out_b(fbr_addr_b), .data_out_c(fbr_addr_c));
tmr_voter_32bit_trip voter19 (.a(njump_addr_a), .b(njump_addr_b), .c(njump_addr_c),
.data_out_a(fjump_addr_a), .data_out_b(fjump_addr_b), .data_out_c(fjump_addr_c));
tmr_voter_32bit_trip voter20 (.a(nalu_result_a), .b(nalu_result_b), .c(nalu_result_c),
.data_out_a(falu_result_a), .data_out_b(falu_result_b), .data_out_c(falu_result_c));
tmr_voter_32bit_trip voter21 (.a(nwr_data_a), .b(nwr_data_b), .c(nwr_data_c),
.data_out_a(fwr_data_a), .data_out_b(fwr_data_b), .data_out_c(fwr_data_c));
tmr_voter_5bit_trip voter22 (.a(nwb_reg_addr_a), .b(nwb_reg_addr_b), .c(nwb_reg_addr_c),
.data_out_a(fwb_reg_addr_a), .data_out_b(fwb_reg_addr_b), .data_out_c(fwb_reg_addr_c));
```

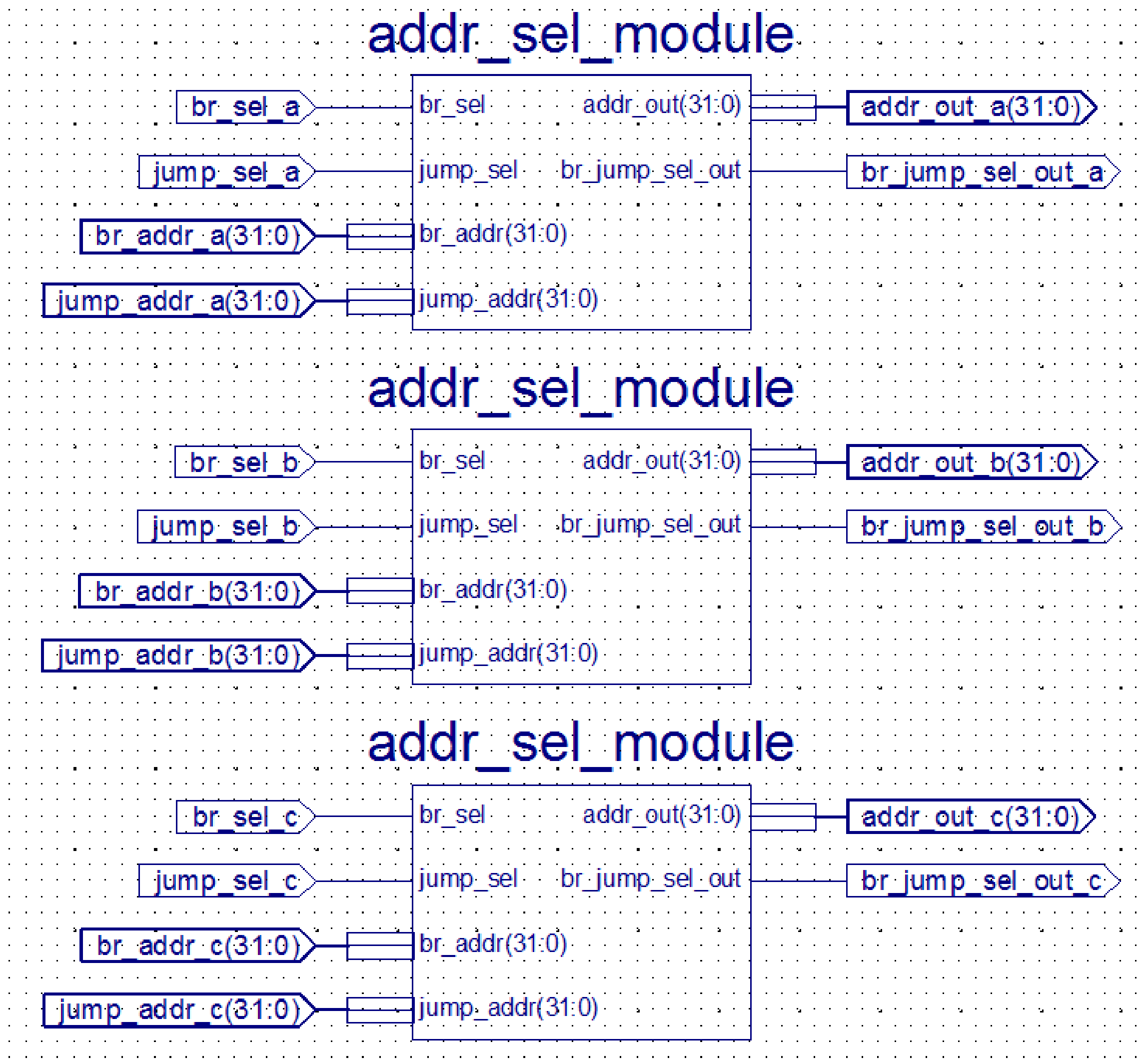
(continued on next page)

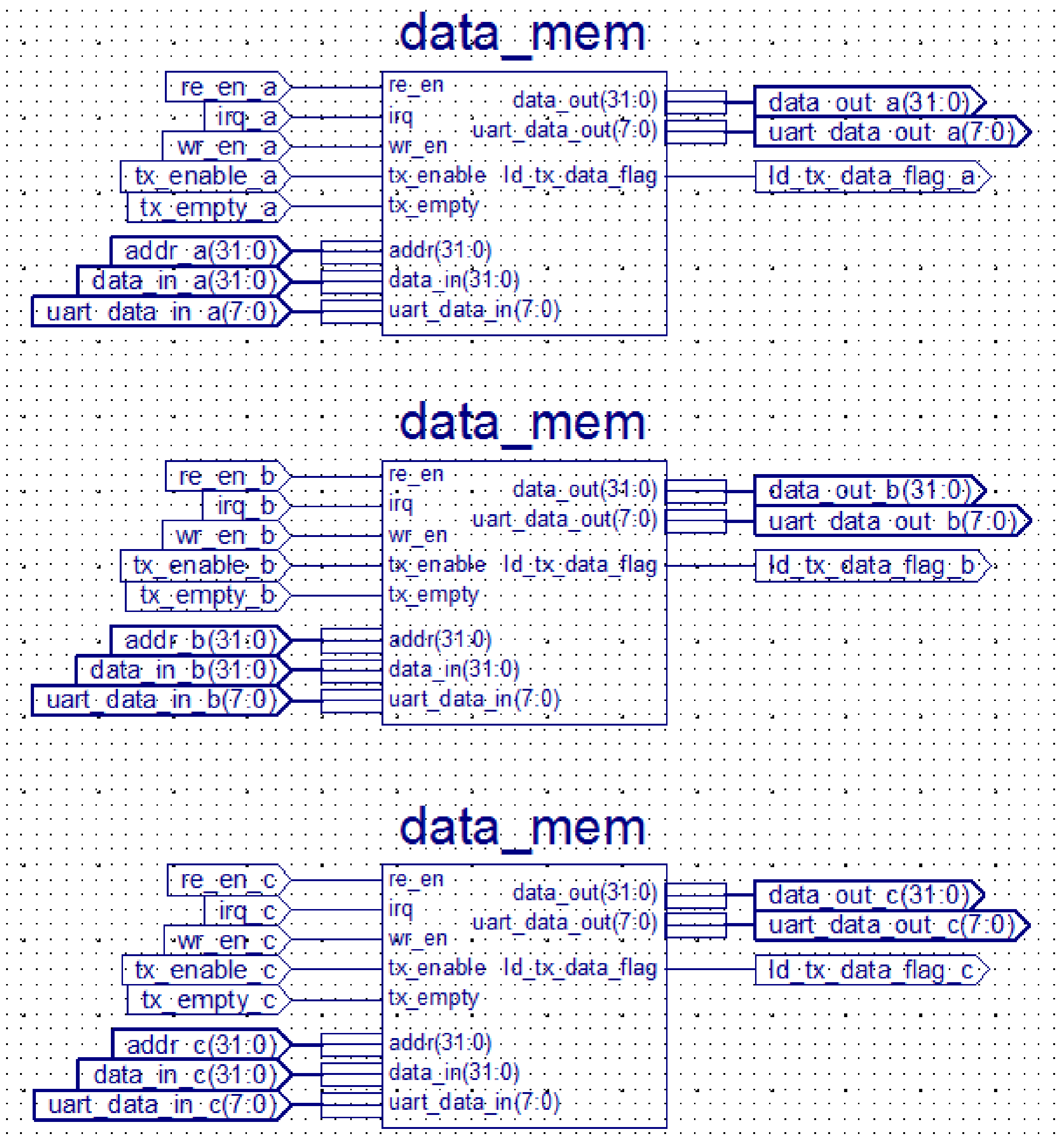
```
    // Pipeline register declaration
    EX_MEM_reg reg_a (.clk(clk), .memread(memread_a), .memwrite(memwrite_a),
    .memtoreg(memtoreg_a), .regwrite(regwrite_a), .beq(beq_a), .bne(bne_a),
    .blz(blz_a), .bgz(bgz_a), .blez(blez_a), .bgez(bgez_a), .seq(seq_a),
    .sne(sne_a), .slz(slz_a), .sgz(sgz_a), .slez(slez_a), .sgez(sgez_a),
    .irqp(irqp_a), .jump(jump_a), .b_addr(br_addr_a), .jump_addr(jump_addr_a),
    .alu_result(alu_result_a), .wr_data(wr_data_a), .wb_reg_addr(wb_reg_addr_a),
    .fmemread(nmemread_a), .fmemwrite(nmemwrite_a), .fmemtoreg(nmemtoreg_a),
    .fregwrite(nregwrite_a), .fbeq(nbeq_a), .fbne(nbne_a), .fblz(nblz_a),
    .fbgz(nbgz_a), .fblez(nblez_a), .fbgez(nbgez_a), .fseq(nseq_a), .fsne(nsne_a),
    .fslz(nslz_a), .fsgz(nsgz_a), .fslez(nslez_a), .fsgez(nsgez_a), .fjump(njump_a),
    .firqp(nirqp_a), .fb_addr(nbr_addr_a), .fjump_addr(njump_addr_a),
    .falu_result(nalu_result_a), .fwr_data(nwr_data_a) , .fwb_reg_addr(nwb_reg_addr_a));

    EX_MEM_reg reg_b (.clk(clk), .memread(memread_b), .memwrite(memwrite_b),
    .memtoreg(memtoreg_b), .regwrite(regwrite_b), .beq(beq_b), .bne(bne_b),
    .blz(blz_b), .bgz(bgz_b), .blez(blez_b), .bgez(bgez_b), .seq(seq_b),
    .sne(sne_b), .slz(slz_b), .sgz(sgz_b), .slez(slez_b), .sgez(sgez_b),
    .irqp(irqp_b), .jump(jump_b), .b_addr(br_addr_b), .jump_addr(jump_addr_b),
    .alu_result(alu_result_b), .wr_data(wr_data_b), .wb_reg_addr(wb_reg_addr_b),
    .fmemread(nmemread_b), .fmemwrite(nmemwrite_b), .fmemtoreg(nmemtoreg_b),
    .fregwrite(nregwrite_b), .fbeq(nbeq_b), .fbne(nbne_b), .fblz(nblz_b),
    .fbgz(nbgz_b), .fblez(nblez_b), .fbgez(nbgez_b), .fseq(nseq_b), .fsne(nsne_b),
    .fslz(nslz_b), .fsgz(nsgz_b), .fslez(nslez_b), .fsgez(nsgez_b), .fjump(njump_b),
    .firqp(nirqp_b), .fb_addr(nbr_addr_b), .fjump_addr(njump_addr_b),
    .falu_result(nalu_result_b), .fwr_data(nwr_data_b) , .fwb_reg_addr(nwb_reg_addr_b));

    EX_MEM_reg reg_c (.clk(clk), .memread(memread_c), .memwrite(memwrite_c),
    .memtoreg(memtoreg_c), .regwrite(regwrite_c), .beq(beq_c), .bne(bne_c),
    .blz(blz_c), .bgz(bgz_c), .blez(blez_c), .bgez(bgez_c), .seq(seq_c),
    .sne(sne_c), .slz(slz_c), .sgz(sgz_c), .slez(slez_c), .sgez(sgez_c),
    .irqp(irqp_c), .jump(jump_c), .b_addr(br_addr_c), .jump_addr(jump_addr_c),
    .alu_result(alu_result_c), .wr_data(wr_data_c), .wb_reg_addr(wb_reg_addr_c),
    .fmemread(nmemread_c), .fmemwrite(nmemwrite_c), .fmemtoreg(nmemtoreg_c),
    .fregwrite(nregwrite_c), .fbeq(nbeq_c), .fbne(nbne_c), .fblz(nblz_c),
    .fbgz(nbgz_c), .fblez(nblez_c), .fbgez(nbgez_c), .fseq(nseq_c), .fsne(nsne_c),
    .fslz(nslz_c), .fsgz(nsgz_c), .fslez(nslez_c), .fsgez(nsgez_c), .fjump(njump_c),
    .firqp(nirqp_c), .fb_addr(nbr_addr_c), .fjump_addr(njump_addr_c),
    .falu_result(nalu_result_c), .fwr_data(nwr_data_c) , .fwb_reg_addr(nwb_reg_addr_c));

endmodule
```

I. MEM STAGE


buf_5bit_trip
addr_sel_module_trip
branch_multiplexer_trip
data_mem_trip
buf_32bit_tmr


branch_multiplexer
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
seq_a
sne_a
slz_a
sgz_a
slez_a
sgez_a
beq
bne
blz
bgz
blez
bgez
seq
sne
slz
sgz
slez
sgez
branch
branch_a
branch_multiplexer
beq_b
bne_b
blz_b
bgz_b
blez_b
bgez_b
seq_b
sne_b
slz_b
sgz_b
slez_b
sgez_b
beq
bne
blz
bgz
blez
bgez
seq
sne
slz
sgz
slez
sgez
branch
branch_b
branch_multiplexer
beq_c
bne_c
blz_c
bgz_c
blez_c
bgez_c
seq_c
sne_c
slz_c
sgz_c
slez_c
sgez_c
beq
bne
blz
bgz
blez
bgez
seq
sne
slz
sgz
slez
sgez
branch
branch_c


addr_sel_module
br_sel_a
jump_sel_a
br_addr_a(31:0)
jump_addr_a(31:0)
br_sel
jump_sel
br_addr(31:0)
jump_addr(31:0)
addr_out(31:0)
br_jump_sel_out
addr_out_a(31:0)
br_jump_sel_out_a
addr_sel_module
br_sel_b
jump_sel_b
br_addr_b(31:0)
jump_addr_b(31:0)
br_sel
jump_sel
br_addr(31:0)
jump_addr(31:0)
addr_out(31:0)
br_jump_sel_out
addr_out_b(31:0)
br_jump_sel_out_b
addr_sel_module
br_sel_c
jump_sel_c
br_addr_c(31:0)
jump_addr_c(31:0)
br_sel
jump_sel
br_addr(31:0)
jump_addr(31:0)
addr_out(31:0)
br_jump_sel_out
addr_out_c(31:0)
br_jump_sel_out_c


data_mem
re_en_a
irq_a
wr_en_a
tx_enable_a
tx_empty_a
addr_a(31:0)
data_in_a(31:0)
uart_data_in_a(7:0)
re_en
irq
wr_en
tx_enable
tx_empty
addr(31:0)
data_in(31:0)
uart_data_in(7:0)
data_out(31:0)
uart_data_out(7:0)
ld_tx_data_flag
data_out_a(31:0)
uart_data_out_a(7:0)
ld_tx_data_flag_a
data_mem
re_en_b
irq_b
wr_en_b
tx_enable_b
tx_empty_b
addr_b(31:0)
data_in_b(31:0)
uart_data_in_b(7:0)
re_en
irq
wr_en
tx_enable
tx_empty
addr(31:0)
data_in(31:0)
uart_data_in(7:0)
data_out(31:0)
uart_data_out(7:0)
ld_tx_data_flag
data_out_b(31:0)
uart_data_out_b(7:0)
ld_tx_data_flag_b
data_mem
re_en_c
irq_c
wr_en_c
tx_enable_c
tx_empty_c
addr_c(31:0)
data_in_c(31:0)
uart_data_in_c(7:0)
re_en
irq
wr_en
tx_enable
tx_empty
addr(31:0)
data_in(31:0)
uart_data_in(7:0)
data_out(31:0)
uart_data_out(7:0)
ld_tx_data_flag
data_out_c(31:0)
uart_data_out_c(7:0)
ld_tx_data_flag_c

```
module pc_select_module_trip3(br_jump_sel_a, br_jump_sel_b, br_jump_sel_c,
   pc4_addr_a, pc4_addr_b, pc4_addr_c, irq_a, irq_b, irq_c, ack_a, ack_b, ack_c,
   br_jump_addr_a, br_jump_addr_b, br_jump_addr_c, irqp_a, irqp_b, irqp_c, npc_a,
   npc_b, npc_c);

   input br_jump_sel_a, br_jump_sel_b, br_jump_sel_c;
   input irq_a, irq_b, irq_c;
   input [31:0] br_jump_addr_a, br_jump_addr_b, br_jump_addr_c;
   input [31:0] pc4_addr_a, pc4_addr_b, pc4_addr_c;

   output [31:0] npc_a, npc_b, npc_c;
   output ack_a, ack_b, ack_c;
   output irqp_a, irqp_b, irqp_c;

   pc_select_module pc_sel_mod_a (.br_jump_sel(br_jump_sel_a), .pc4_addr(pc4_addr_a),
      .irq(irq_a), .ack(ack_a), .br_jump_addr(br_jump_addr_a), .irqp(irqp_a),
      .npc(npc_a));

   pc_select_module pc_sel_mod_b (.br_jump_sel(br_jump_sel_b), .pc4_addr(pc4_addr_b),
      .irq(irq_b), .ack(ack_b), .br_jump_addr(br_jump_addr_b), .irqp(irqp_b),
      .npc(npc_b));

   pc_select_module pc_sel_mod_c (.br_jump_sel(br_jump_sel_c), .pc4_addr(pc4_addr_c),
      .irq(irq_c), .ack(ack_c), .br_jump_addr(br_jump_addr_c), .irqp(irqp_c),
      .npc(npc_c));


endmodule
```

1. Branch Multiplexer (branch_multiplexer)

```
module branch_multiplexer(beq, bne, blz, bgz, blez, bgez, seq, sne, slz, sgz, slez,
   sgez, branch);

   input beq, bne, blz, bgz, blez, bgez, seq, sne, slz, sgz, slez, sgez;
   output reg branch;

   initial
      branch = 0;

   // Branch conditions are met when a pair of branch flags and comparison flags
   // are both high.  This only triggers the branch/jump selector bit for the
   // address selector module, which will forward the result of this module and the
   // branch address calculated in the EX stage.
   always @(beq or bne or blz or bgz or blez or bgez or seq or sne or slz or sgz or
      slez or sgez)
      branch = (beq & seq) | (bne & sne) | (blz & slz) | (bgz & sgz) |
      (blez & slez) | (bgez & sgez);

endmodule
```

2. Address Selector Module (addr_sel_module)

```
module addr_sel_module (br_addr, jump_addr, br_sel, jump_sel, addr_out,
   br_jump_sel_out);

   input br_sel, jump_sel; // Branch and jump selector bits set during the ID stage.
   input [31:0] br_addr, jump_addr; // Branch and jump addresses set during the IF
                                    // and EX stages, respectively.
   output reg [31:0] addr_out; // Output address
   output reg br_jump_sel_out; // Output flag

   initial
      begin
         addr_out = 0;
         br_jump_sel_out = 0;
      end

   always @(br_addr or jump_addr or br_sel or jump_sel)
      begin
         if (br_sel & ~jump_sel) // Branch criterion met
            begin
               addr_out = br_addr; // Forward branch address to the PC selector module
               br_jump_sel_out = 1; // Raise the branch/jump selector flag
            end
         else if (~br_sel & jump_sel) // Jump criterion met
            begin
               addr_out = jump_addr; // Forward jump address to the PC selector module
               br_jump_sel_out = 1; // Raise the branch/jump selector flag
            end
         else
            br_jump_sel_out = 0; // No branch or jump criterion have been met, the next
                                 // instruction address should be the normal PC + 4.
      end

endmodule
```

3. Data Memory (data_mem)

```
module data_mem(addr, data_in, uart_data_in, wr_en, re_en, irq, tx_enable, tx_empty,
   ld_tx_data_flag, data_out, uart_data_out);

   input [31:0] addr, data_in; // Input data and address
   input [7:0] uart_data_in; // UART write address
   input re_en, wr_en, irq, tx_enable, tx_empty; // Flags and control bits
   output reg [31:0] data_out; // Data read from memory
   output reg [7:0] uart_data_out; // Data read from special UART address
   output reg ld_tx_data_flag; // Flag to initiate UART transmit

   reg [7:0] data_byte0, data_byte1, data_byte2, data_byte3; // Byte components
   reg [7:0] data_memory_module [0:44999]; // Memory data structure
   reg [31:0] uart_wr_addr; // Current UART write address
   reg [31:0] data_assemble; // Assembly of byte-wide components for load word instructions

   always @(addr or data_in or uart_data_in or wr_en or re_en or irq or tx_enable or
      tx_empty)
      begin
         ld_tx_data_flag = 0; // Ensure the UART transmit flag is low if there is no irqp
         if (~irq)
            begin
               if (wr_en && ~re_en) // Memory write operation
                  begin
                     // If the write address is in the UART transmit block, ensure the entire
                     // word is written into the last four addresses.
```

(continued on next page)

```
         if (addr == 44996 | addr == 44997 | addr == 44998 | addr == 44999)
            begin
               data_memory_module [44996] = data_in [31:24];
               data_memory_module [44997] = data_in [23:16];
               data_memory_module [44998] = data_in [15:8];
               data_memory_module [44999] = data_in [7:0];
            end
         // The remaining if/else statements prevent data from being written
         // into the block of address (22496 - 22499) where data memory stores
         // the current address for UART writes.  If an instruction is
         // attempting to write outside the bounds of its address space, the
         // most significant bits that fit in the allocated space will be written.
         else if (addr == 22495)
            data_memory_module [addr] = data_in [31:24];
         else if (addr == 22494)
            begin
               data_memory_module [addr] = data_in [31:24];
               data_memory_module [addr + 1] = data_in [23:16];
            end
         else if (addr == 22493)
            begin
               data_memory_module [addr] = data_in [31:24];
               data_memory_module [addr + 1] = data_in [23:16];
               data_memory_module [addr + 2] = data_in [15:8];
            end
         else if (addr < 22493)
            begin
               data_memory_module [addr] = data_in [31:24];
               data_memory_module [addr + 1] = data_in [23:16];
               data_memory_module [addr + 2] = data_in [15:8];
               data_memory_module [addr + 3] = data_in [7:0];
            end
      end
   if (~wr_en && re_en) // Memory read operation
      begin
         // If a read operation occurs where the address specified is in the
         // UART transmit block, only a single byte of data from the very last
         // address in memory will be sent to the UART for transmission.  The
         // memory addresses will then be sent to the register specified in
         // the load instruction, with the least significant byte being the
         // one transmitted by the UART.
         if (addr == 44996 | addr == 44997 | addr == 44998 | addr == 44999)
            begin
               uart_data_out = data_memory_module [44999]; // Send the least
                                                           // significant byte to
                                                           // the UART.
               ld_tx_data_flag = 1; // Flag to trigger UART transmission
               // Standard operation to write an entire word to a register as
               // part of the load word instruction.
               data_byte0 = data_memory_module [44996];
               data_byte1 = data_memory_module [44997];
               data_byte2 = data_memory_module [44998];
               data_byte3 = data_memory_module [44999];
               data_out = {data_byte0, data_byte1, data_byte2, data_byte3};
            end
         // Memory in the address space, 22499 < addr < 44996, is reserved for the
         // UART and can only be written in single bytes.  Any load word instruction
         // to this memory space will return data as the least significant byte of
         // the register.
         else if (addr > 22499 & addr < 44996)
            begin
               data_byte0 = 8'b00000000;
               data_byte1 = 8'b00000000;
               data_byte2 = 8'b00000000;
               data_byte3 = data_memory_module [addr];
               data_out = {data_byte0, data_byte1, data_byte2, data_byte3};
            end
```
(continued on next page)

```
                // Memory in the address space, addr < 22497, can read an entire word into
                // the register.  Read operations referencing 22497 to 22499 will read
                // as much data as allocated in the memory space without reading into the
                // UART data.
                else if (addr >= 0 & addr < 22497)
                   begin
                      data_byte0 = data_memory_module [addr];
                      data_byte1 = data_memory_module [addr + 1];
                      data_byte2 = data_memory_module [addr + 2];
                      data_byte3 = data_memory_module [addr + 3];
                      data_out = {data_byte0, data_byte1, data_byte2, data_byte3};
                   end
                else if (addr == 22497)
                   begin
                      data_byte0 = 8'b00000000;
                      data_byte1 = data_memory_module [addr];
                      data_byte2 = data_memory_module [addr + 1];
                      data_byte3 = data_memory_module [addr + 2];
                      data_out = {data_byte0, data_byte1, data_byte2, data_byte3};
                   end
                else if (addr == 22498)
                   begin
                      data_byte0 = 8'b00000000;
                      data_byte1 = 8'b00000000;
                      data_byte2 = data_memory_module [addr];
                      data_byte3 = data_memory_module [addr + 1];
                      data_out = {data_byte0, data_byte1, data_byte2, data_byte3};
                   end
                else if (addr == 22499)
                   begin
                      data_byte0 = 8'b00000000;
                      data_byte1 = 8'b00000000;
                      data_byte2 = 8'b00000000;
                      data_byte3 = data_memory_module [addr];
                      data_out = {data_byte0, data_byte1, data_byte2, data_byte3};
                   end
             end
       end
    else if (irq & ~tx_enable & tx_empty) // Interrupt request present
       begin
          // Temporary bytes are assigned to the current write address in UART memory.
          // The temporary bytes are then concatenated to obtain a complete memory
          // address that is then used as a reference in the memory assignment.
          data_byte0 = data_memory_module [22496];
          data_byte1 = data_memory_module [22497];
          data_byte2 = data_memory_module [22498];
          data_byte3 = data_memory_module [22499];
          uart_wr_addr = {data_byte0, data_byte1, data_byte2, data_byte3};
          data_memory_module [uart_wr_addr] = uart_data_in;
          // In the event the UART reaches its last write address, the address counter
          // wraps around to the first address and updates the placeholder in memory.
          if (uart_wr_addr == 44995)
             begin
                uart_wr_addr = 22500;
                data_byte0 = uart_wr_addr [31:24];
                data_byte1 = uart_wr_addr [23:16];
                data_byte2 = uart_wr_addr [15:8];
                data_byte3 = uart_wr_addr [7:0];
                data_memory_module [22496] = data_byte0;
                data_memory_module [22497] = data_byte1;
                data_memory_module [22498] = data_byte2;
                data_memory_module [22499] = data_byte3;
             end
          // If the UART has not yet reached the end of data memory, the address is
          // incremented by one, and the placeholder address is updated.
          else
             begin
                uart_wr_addr = uart_wr_addr + 1;
                data_byte0 = uart_wr_addr [31:24];
                data_byte1 = uart_wr_addr [23:16];
                data_byte2 = uart_wr_addr [15:8];
                data_byte3 = uart_wr_addr [7:0];
```

```
                        data_memory_module [22496] = data_byte0;
                        data_memory_module [22497] = data_byte1;
                        data_memory_module [22498] = data_byte2;
                        data_memory_module [22499] = data_byte3;
                     end
               end
         end

   initial // Data memory is initialized from a file.
      begin
         $readmemb ("payload_processor_data_memory_init.txt", data_memory_module);
      end

endmodule
```

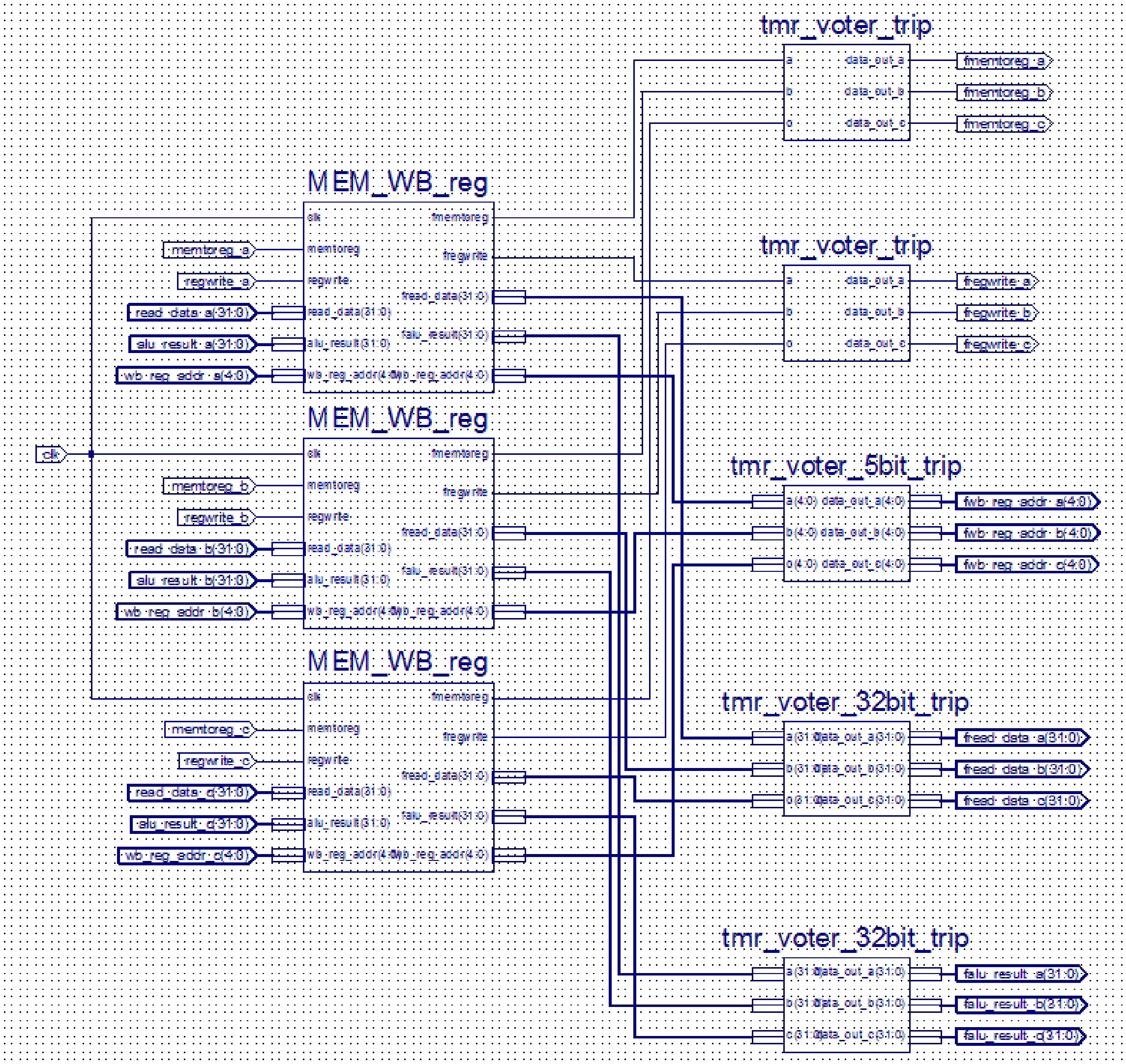
4. PC Selector Module

```
module pc_select_module(br_jump_sel, irq, ack, br_jump_addr, pc4_addr, irqp, npc);

   parameter isr_start_addr = 44980;
   parameter isr_end_addr = 44996;

   input wire br_jump_sel, irq;
   input wire [31:0] br_jump_addr, pc4_addr;
   output reg [31:0] npc;
   output reg ack, irqp;
   reg irq_processing;
   reg [31:0] irq_return_addr;
   reg [31:0] curr_pc;

   initial
      begin
         npc = 4;
         ack = 0;
         irqp = 0;
         irq_processing = 0;
         irq_return_addr = 44999;
         curr_pc = 0;
      end

   always @(posedge br_jump_sel)
      begin
         if (irq_processing)
            irq_return_addr = br_jump_addr;
      end

   always @(br_jump_sel or irq or br_jump_addr or pc4_addr)
      begin
         // An interrupt request has occurred, and the system is not currently
         // servicing any interrupts.
         if (irq)
            begin
               ack = 1; // Acknowledges IRQ of UART and flags the UART to place
                        // data into the appropriate MEM stage inputs.
               irqp = 0;
               irq_processing = 1; // Places processor in an interrupt state
               // If the branch/jump address selector is low, there is no branch
               // or jump instruction about to be processed.  The return address
               // should be PC + 4.
               if (~br_jump_sel)
                  begin
                     irq_return_addr = pc4_addr;
                     curr_pc = isr_start_addr;
                     npc = curr_pc; // Set the new PC for the ISR
                     curr_pc = curr_pc + 4; // Set the ISR PC to the
                                            // first ISR address.
                  end
               // If the branch/jump address selector is high, there is a branch
               // or jump instruction about to be processed.  The return address
               // should be set to this value instead of PC + 4.
               else
                  begin
                     irq_return_addr = br_jump_addr;
                     curr_pc = isr_start_addr;
                     npc = curr_pc; // Set the new PC for the ISR
                     curr_pc = curr_pc + 4; // Set the ISR PC to the
                                            // first ISR address.
                  end

            end
```

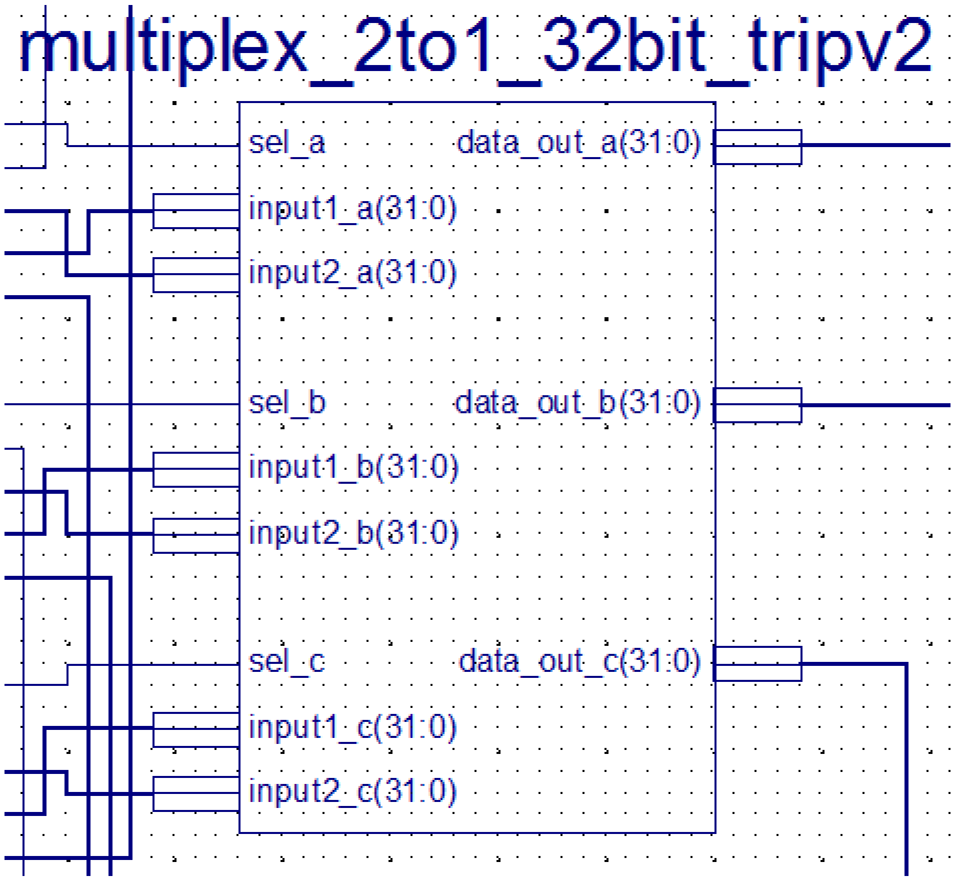
(continued on next page)

```
        else
           begin
              if (irq_processing)
                 begin
                    ack = 0;
                    // If the next PC value is still less than the end address of the
                    // ISR, the system is still busy performing the ISR on its
                    // current UART data.  The next PC address should be the
                    // instruction immediately following the current one.
                    if (pc4_addr <= isr_end_addr)
                       begin
                          curr_pc = pc4_addr;
                          npc = curr_pc; // Update to the next PC address
                          // Trigger the IRQP flag bit during the final ISR instruction
                          if (pc4_addr == isr_end_addr)
                             irqp = 1;
                       end
                    // If the next PC value is greater than the end address of the ISR,
                    // the system has completed its ISR and the next PC address should
                    // be the previously saved return address.
                    else
                       begin
                          npc = irq_return_addr;
                          curr_pc = irq_return_addr;
                          irq_processing = 0;
                          irqp = 0;
                       end
                 end
              // If no interrupt requests are present, and the system is not servicing
              // an interrupt, the next PC value is selected based on the state of the
              // br_jump_sel signal.
              else
                 begin
                    ack = 0;
                    // The br_jump_sel flag is set high, meaning there is a branch or
                    // jump address that must serve as the next PC address.
                    if (br_jump_sel)
                       begin
                          curr_pc = br_jump_addr;
                          npc = curr_pc;
                          curr_pc = curr_pc + 4;
                       end
                    // The br_jump_sel is set low, meaning there is no branch or jump
                    // address, and the next PC value should be PC + 4.
                    else
                       begin
                          // Perform special processing upon return from the
                          if (curr_pc == irq_return_addr)
                             begin
                                curr_pc = curr_pc + 4;
                                npc = curr_pc;
                                irq_return_addr = 44999; // Ensures the if statement is
                                                         // not re-entered during the
                                                         // next clock cycle
                                irqp = 0;
                             end
                          // Standard assignment of the next PC address
                          else
                             begin
                                curr_pc = pc4_addr;
                                npc = curr_pc;
                                irqp = 0;
                             end
                       end
                    irq_processing = 0;
                 end
           end
     end

endmodule
```

J. MEM/WB REGISTER



K. WB STAGE




```
module multiplex_2to1_32bit_tripv2(input1_a, input1_b, input1_c, input2_a,
   input2_b, input2_c, sel_a, sel_b, sel_c, data_out_a, data_out_b, data_out_c);

   input [31:0] input1_a, input1_b, input1_c, input2_a, input2_b, input2_c;
   input sel_a, sel_b, sel_c;

   output [31:0] data_out_a, data_out_b, data_out_c;

   multiplex_2to1_32bitv2 mux1 (.a(input1_a), .b(input2_a), .sel(sel_a),
      .data_out(data_out_a));

   multiplex_2to1_32bitv2 mux2 (.a(input1_b), .b(input2_b), .sel(sel_b),
      .data_out(data_out_b));

   multiplex_2to1_32bitv2 mux3 (.a(input1_c), .b(input2_c), .sel(sel_c),
      .data_out(data_out_c));


endmodule
```

1. Write–Back Multiplexer

```
module multiplex_2to1_32bitv2(a, b, sel, data_out);

   input [31:0] a, b;
   input sel;

   output reg [31:0] data_out;

   always @(a or b or sel)
      begin
         if (~sel)
            data_out = a;
         else
            data_out = b;
      end

endmodule
```

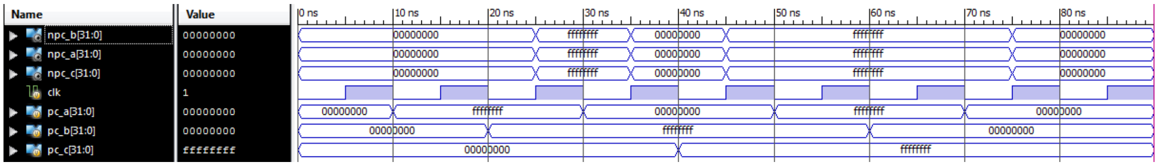
APPENDIX B. TEST BENCHES AND WAVEFORMS

This appendix contains the simulation results for the test program with UART integration. In Section A, all of the pipeline registers and voters are tested to ensure they are capable of correcting SBUs. Similar to the methods proposed in Chapter V, operational verification of the pipeline registers and voters allows the test program to be viewed through the lens of a single member of the triplicated set of processors (processor A).

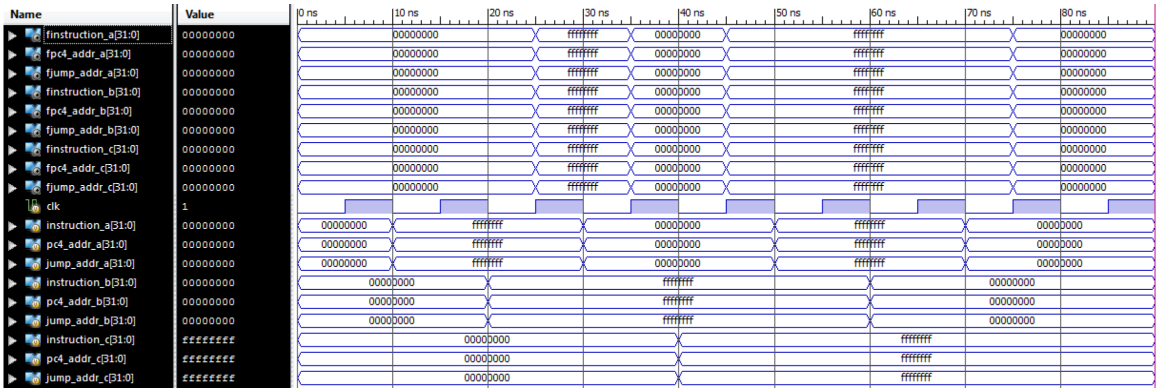
In Sections B – M of this appendix, the test program is verified using the inputs and outputs of each pipeline register and stage. Each section displays a single pipeline stage, pipeline register, or UART input and output signals. Within these sections, the inputs and outputs for that module are displayed consecutively from 0 – 336 ns in increments of 16 ns. These simulation results include the receipt of a byte of data at the UART receive interface that transitions to the ISR and the transmission of a data byte from memory through the UART transmit interface. The test bench used to initiate the simulation is included in Section N.

A. PIPELINE REGISTERS AND VOTING LOGIC

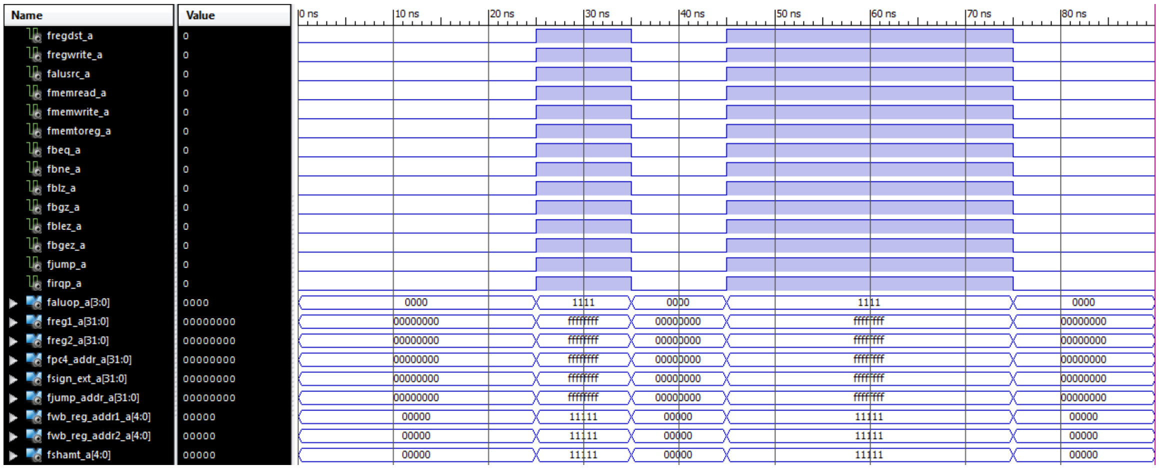
1. PC Register

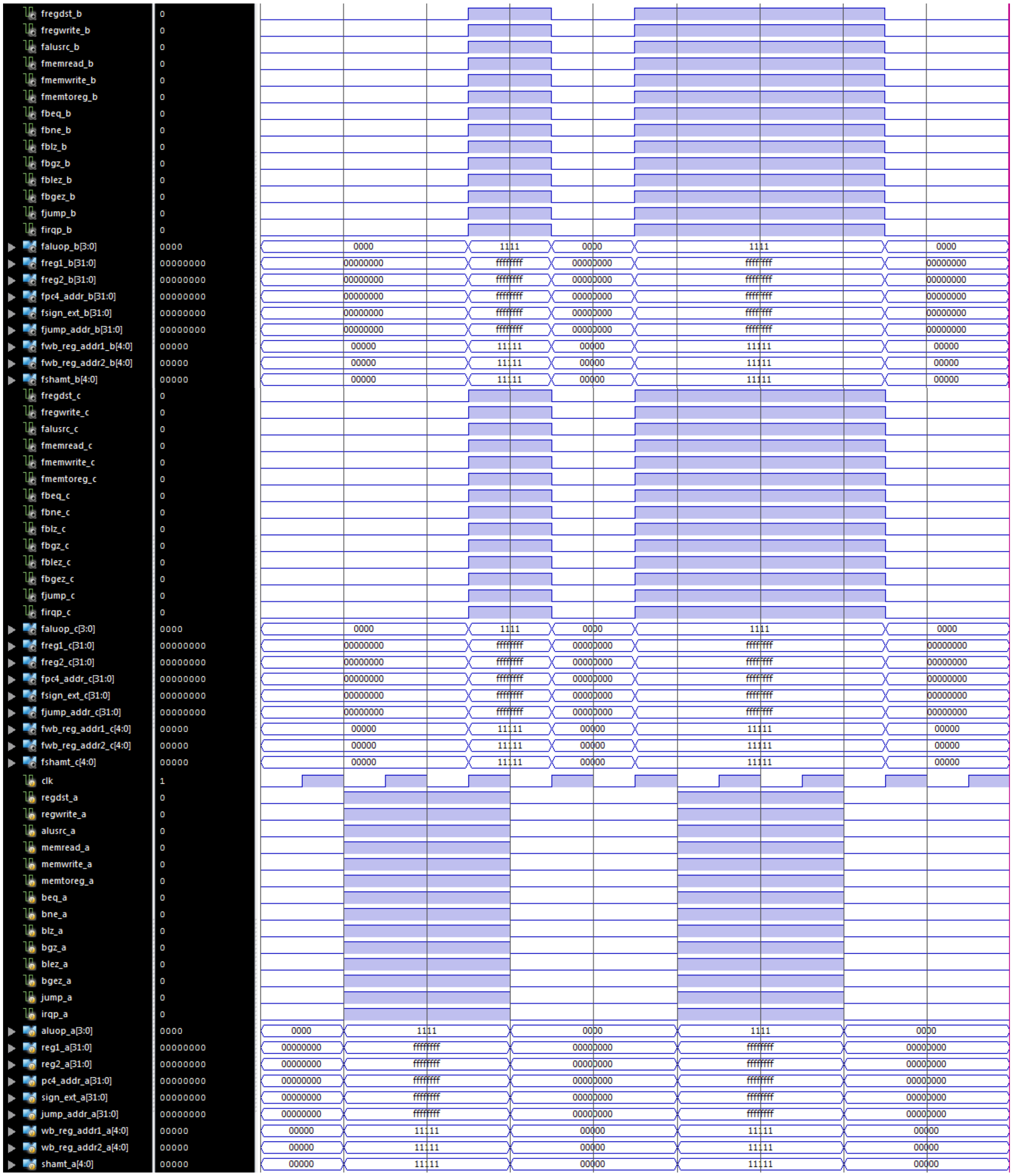


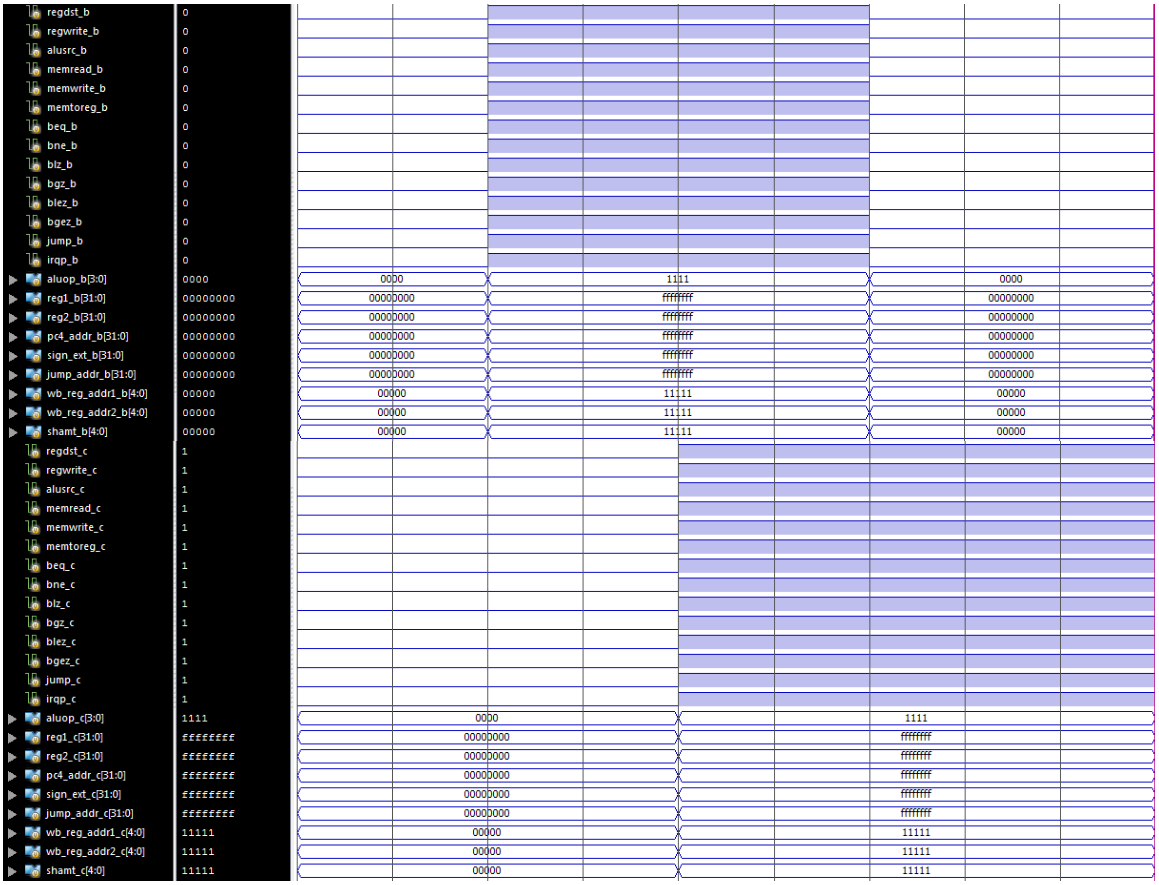
2. IF/ID Register



3. ID/EX Register

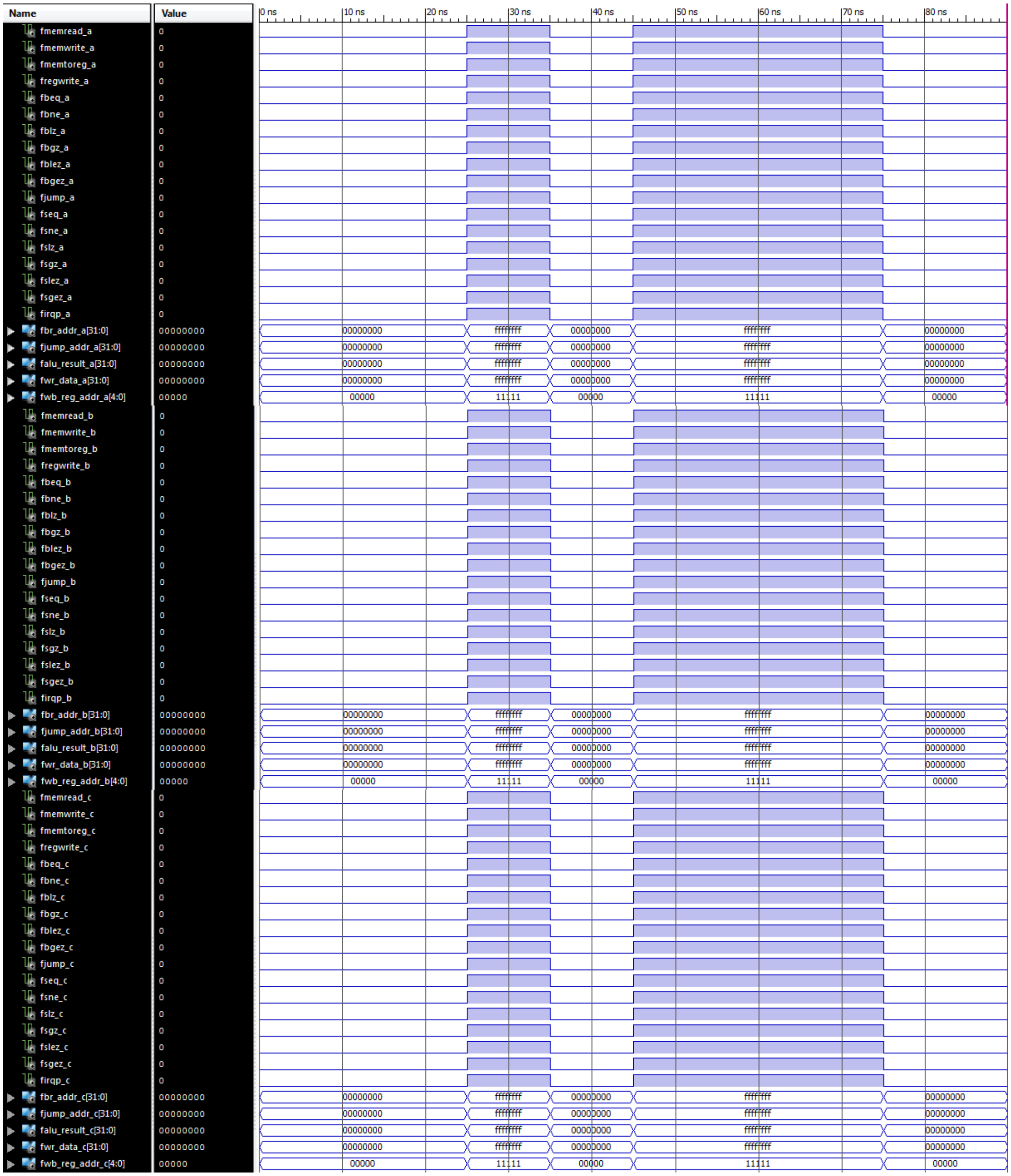


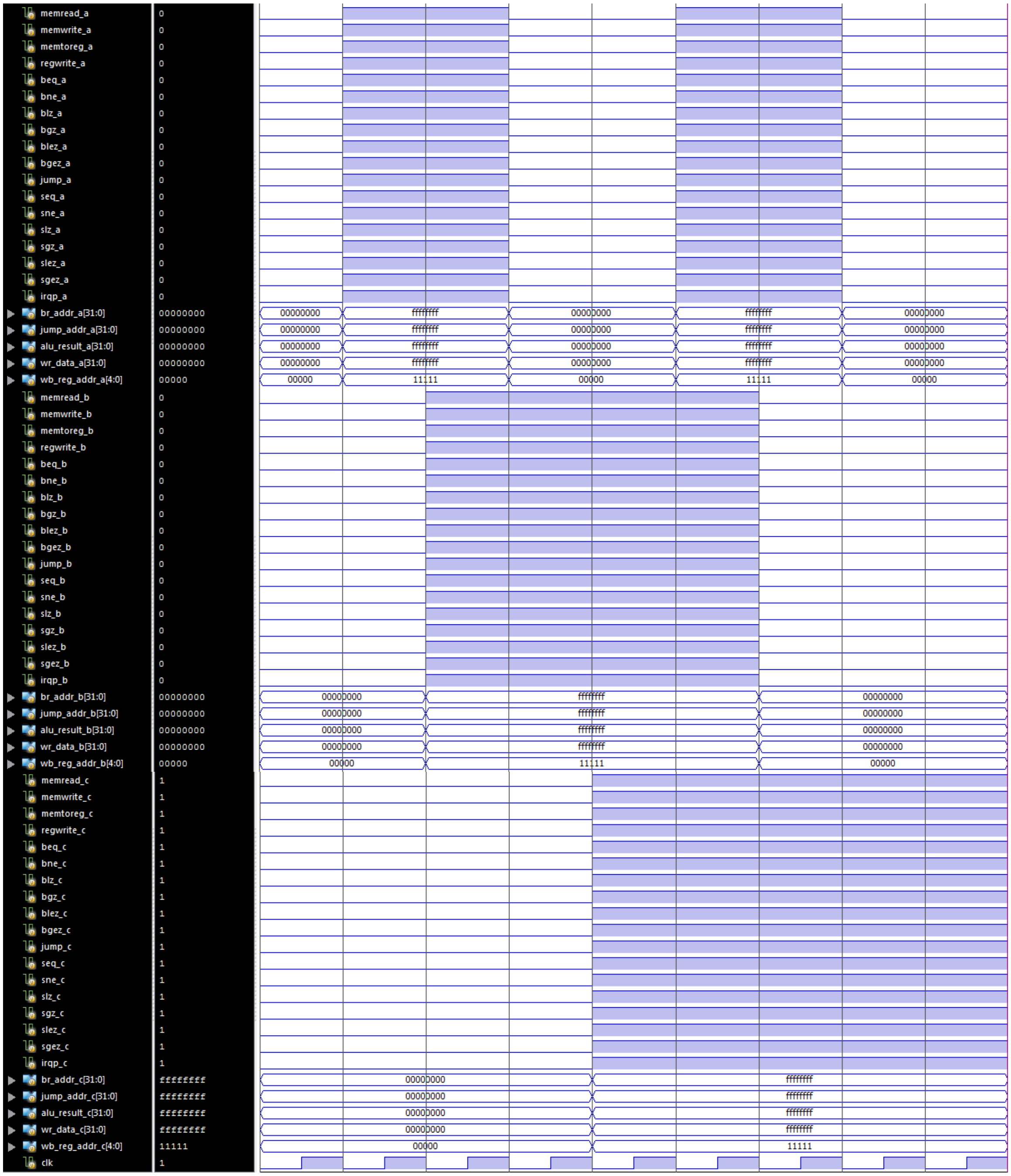




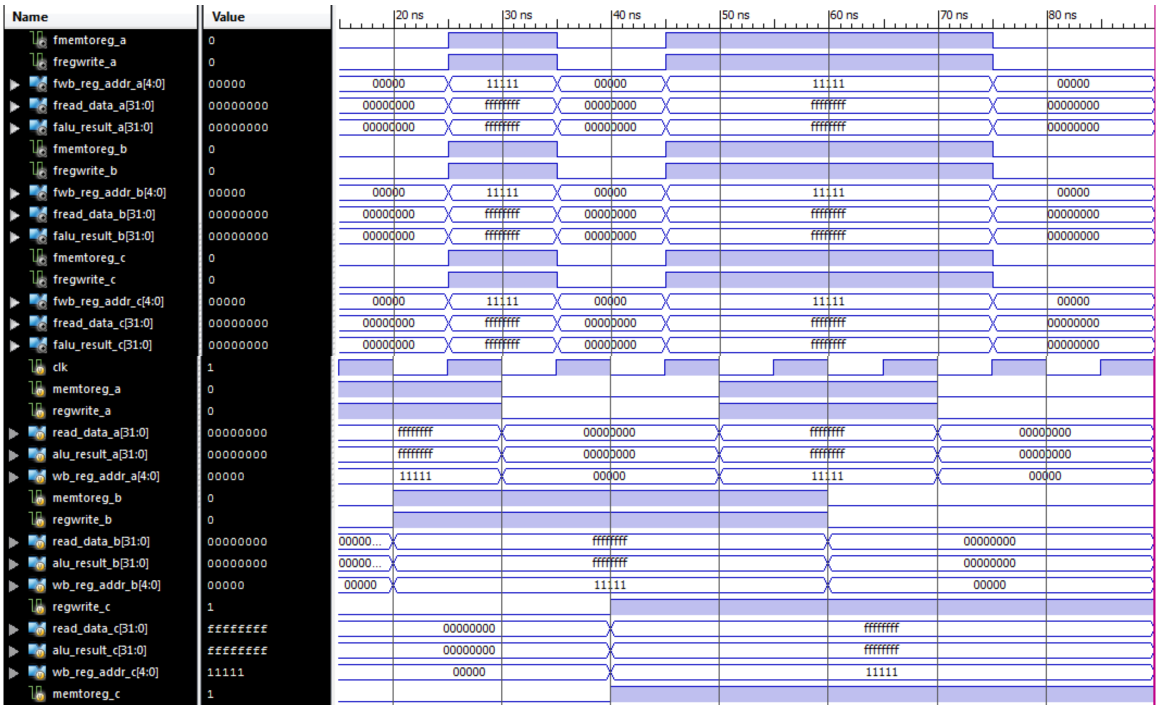
regdst_b
regwrite_b
alusrc_b
memread_b
memwrite_b
memtoreg_b
beq_b
bne_b
blz_b
bgz_b
blez_b
bgez_b
jump_b
irqp_b
aluop_b[3:0]
reg1_b[31:0]
reg2_b[31:0]
pc4_addr_b[31:0]
sign_ext_b[31:0]
jump_addr_b[31:0]
wb_reg_addr1_b[4:0]
wb_reg_addr2_b[4:0]
shamt_b[4:0]
regdst_c
regwrite_c
alusrc_c
memread_c
memwrite_c
memtoreg_c
beq_c
bne_c
blz_c
bgz_c
blez_c
bgez_c
jump_c
irqp_c
aluop_c[3:0]
reg1_c[31:0]
reg2_c[31:0]
pc4_addr_c[31:0]
sign_ext_c[31:0]
jump_addr_c[31:0]
wb_reg_addr1_c[4:0]
wb_reg_addr2_c[4:0]
shamt_c[4:0]

4. EX/MEM Register

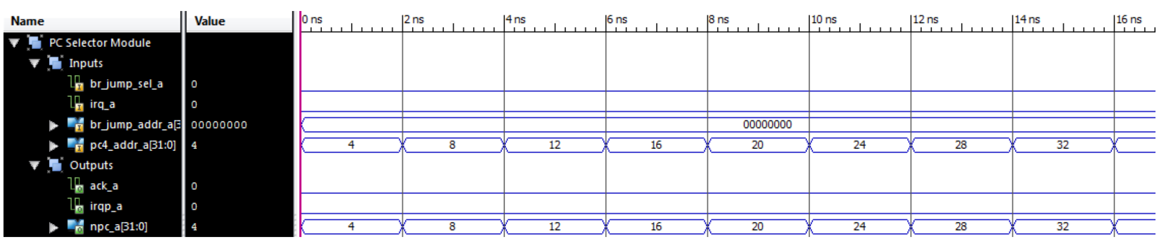


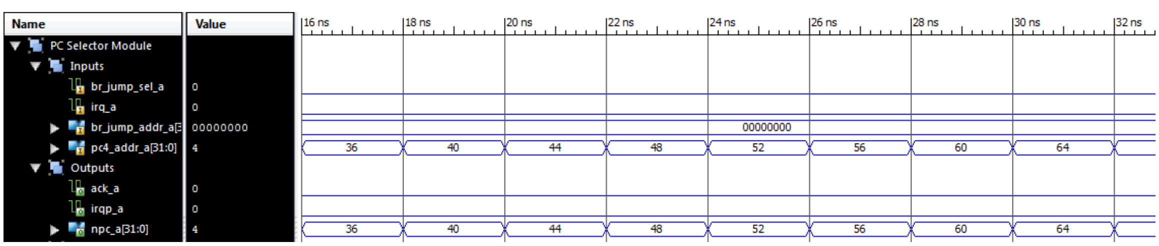

memread_a
memwrite_a
memtoreg_a
regwrite_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
seq_a
sne_a
slz_a
sgz_a
slez_a
sgez_a
irqp_a
br_addr_a[31:0]
jump_addr_a[31:0]
alu_result_a[31:0]
wr_data_a[31:0]
wb_reg_addr_a[4:0]
memread_b
memwrite_b
memtoreg_b
regwrite_b
beq_b
bne_b
blz_b
bgz_b
blez_b
bgez_b
jump_b
seq_b
sne_b
slz_b
sgz_b
slez_b
sgez_b
irqp_b
br_addr_b[31:0]
jump_addr_b[31:0]
alu_result_b[31:0]
wr_data_b[31:0]
wb_reg_addr_b[4:0]
memread_c
memwrite_c
memtoreg_c
regwrite_c
beq_c
bne_c
blz_c
bgz_c
blez_c
bgez_c
jump_c
seq_c
sne_c
slz_c
sgz_c
slez_c
sgez_c
irqp_c
br_addr_c[31:0]
jump_addr_c[31:0]
alu_result_c[31:0]
wr_data_c[31:0]
wb_reg_addr_c[4:0]
clk
00000000
ffffffff
00000
11111

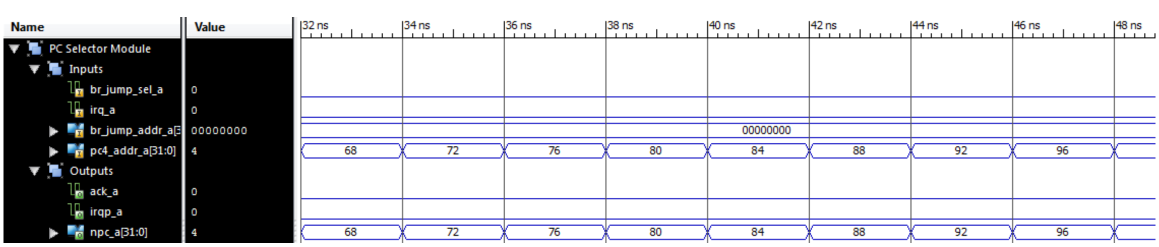
5. MEM/WB Register

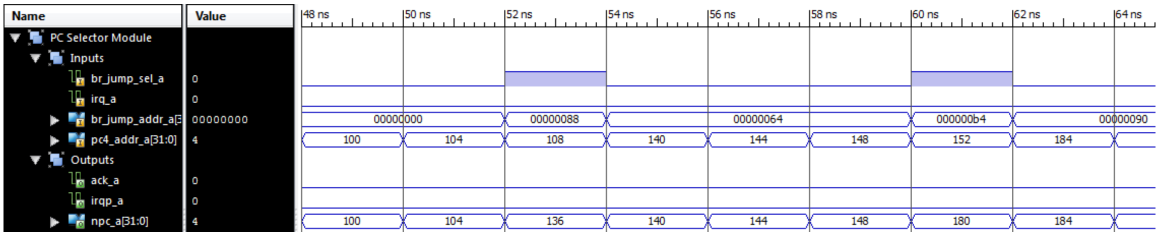


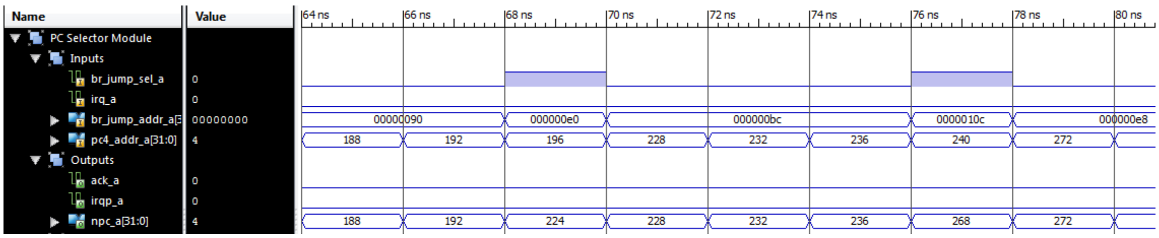
B. PC SELECTOR MODULE

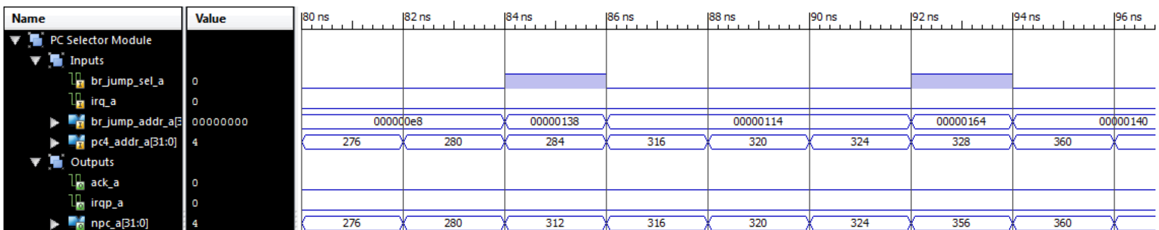


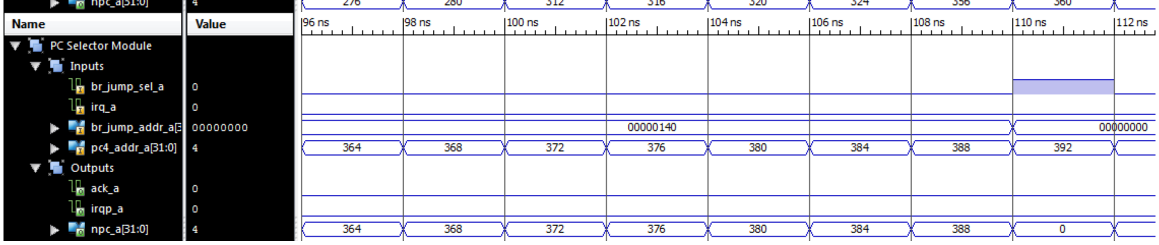


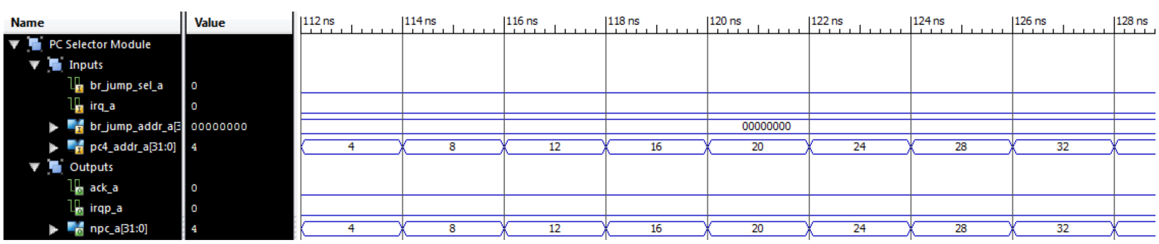


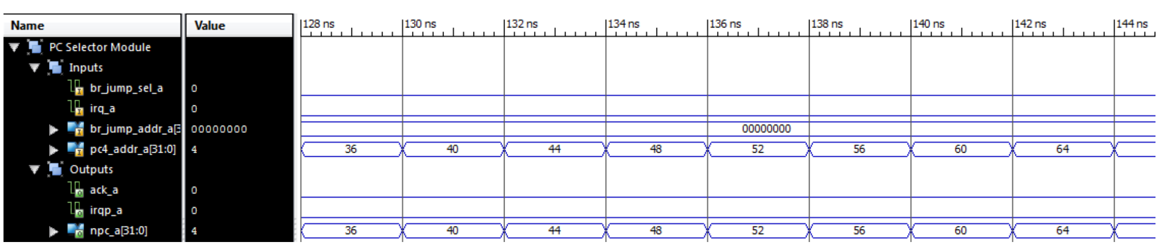

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
48 ns 50 ns 52 ns 54 ns 56 ns 58 ns 60 ns 62 ns 64 ns
00000000 00000088 00000064 000000b4 00000090
100 104 108 140 144 148 152 184
100 104 136 140 144 148 180 184

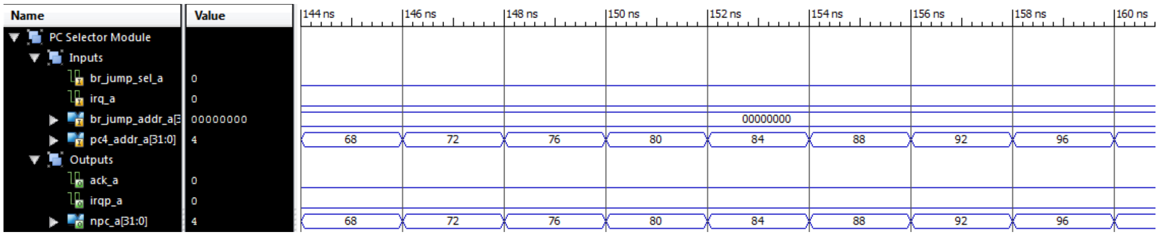

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
64 ns 66 ns 68 ns 70 ns 72 ns 74 ns 76 ns 78 ns 80 ns
00000090 000000e0 000000bc 0000010c 000000e8
188 192 196 228 232 236 240 272
188 192 224 228 232 236 268 272

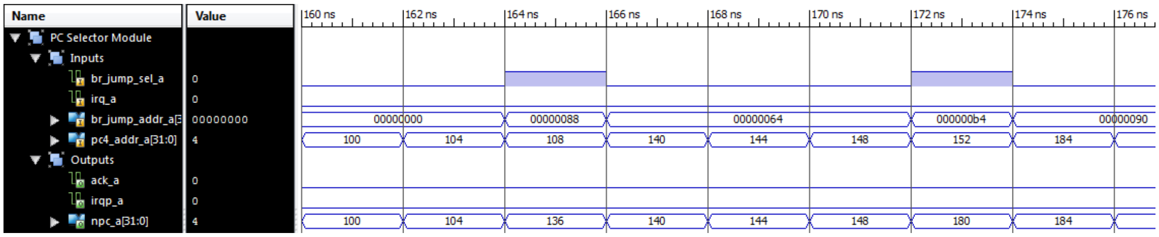

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
80 ns 82 ns 84 ns 86 ns 88 ns 90 ns 92 ns 94 ns 96 ns
000000e8 00000138 00000114 00000164 00000140
276 280 284 316 320 324 328 360
276 280 312 316 320 324 356 360

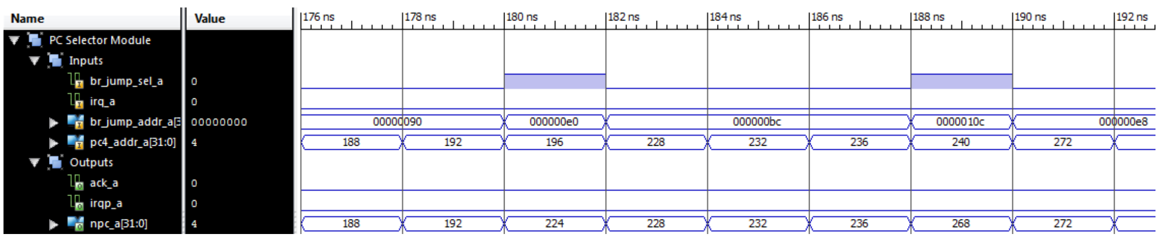

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
96 ns 98 ns 100 ns 102 ns 104 ns 106 ns 108 ns 110 ns 112 ns
00000140 00000000
364 368 372 376 380 384 388 392
364 368 372 376 380 384 388 0

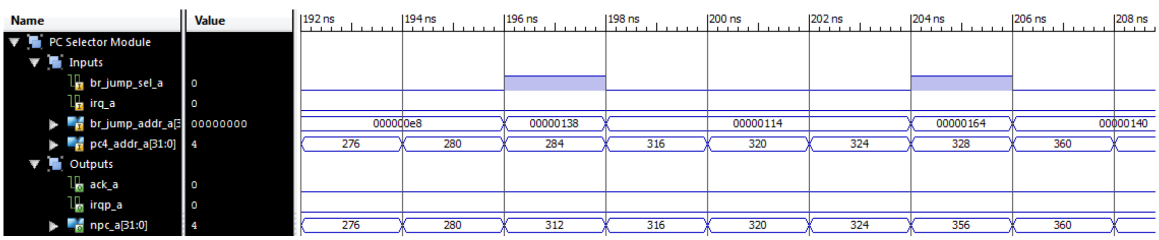

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
112 ns 114 ns 116 ns 118 ns 120 ns 122 ns 124 ns 126 ns 128 ns
00000000
4 8 12 16 20 24 28 32
4 8 12 16 20 24 28 32

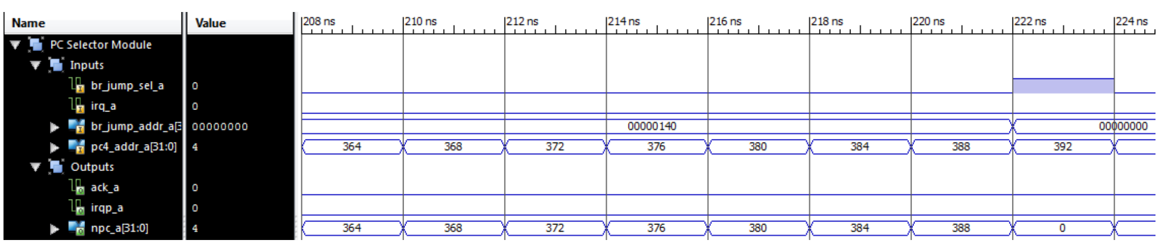

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
128 ns 130 ns 132 ns 134 ns 136 ns 138 ns 140 ns 142 ns 144 ns
00000000
36 40 44 48 52 56 60 64
36 40 44 48 52 56 60 64

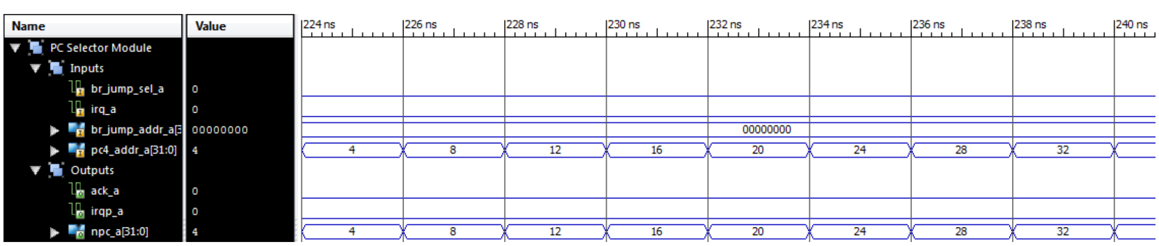

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
144 ns 146 ns 148 ns 150 ns 152 ns 154 ns 156 ns 158 ns 160 ns
00000000
68 72 76 80 84 88 92 96
68 72 76 80 84 88 92 96

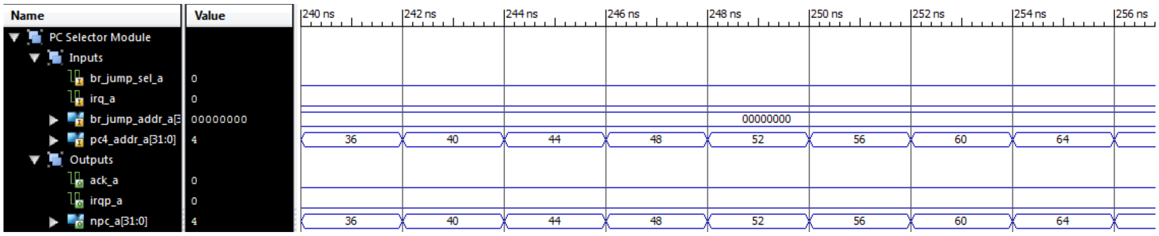

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
160 ns 162 ns 164 ns 166 ns 168 ns 170 ns 172 ns 174 ns 176 ns
00000000 00000088 00000064 000000b4 00000090
100 104 108 140 144 148 152 184
100 104 136 140 144 148 180 184

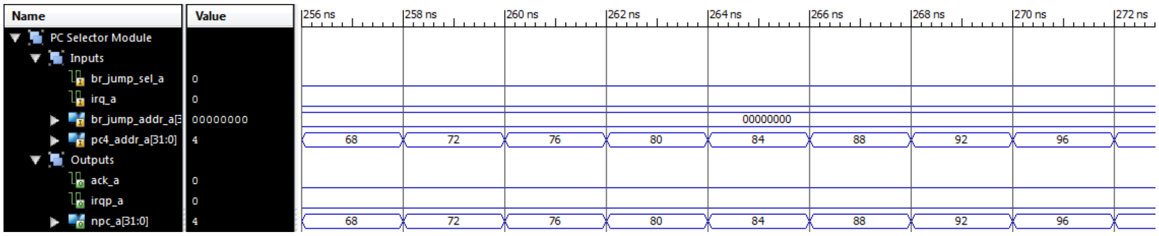

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
176 ns 178 ns 180 ns 182 ns 184 ns 186 ns 188 ns 190 ns 192 ns
00000090 000000e0 000000bc 0000010c 000000e8
188 192 196 228 232 236 240 272
188 192 224 228 232 236 268 272

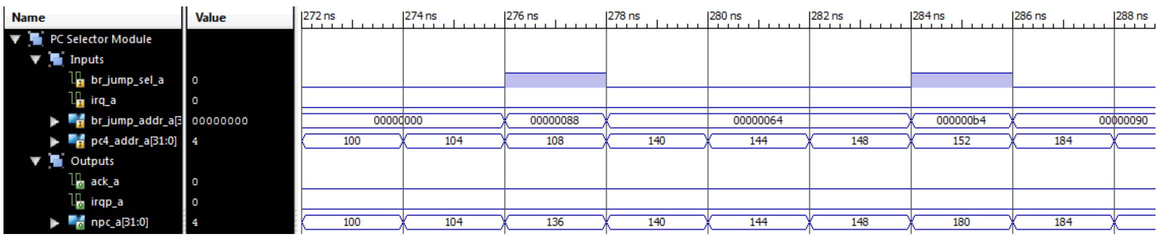

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
192 ns 194 ns 196 ns 198 ns 200 ns 202 ns 204 ns 206 ns 208 ns
000000e8 00000138 00000114 00000164 00000140
276 280 284 316 320 324 328 360
276 280 312 316 320 324 356 360

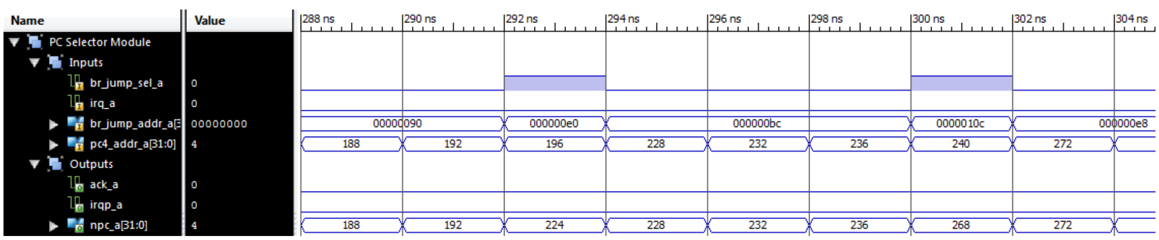

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
208 ns 210 ns 212 ns 214 ns 216 ns 218 ns 220 ns 222 ns 224 ns
00000140 00000000
364 368 372 376 380 384 388 392
364 368 372 376 380 384 388 0

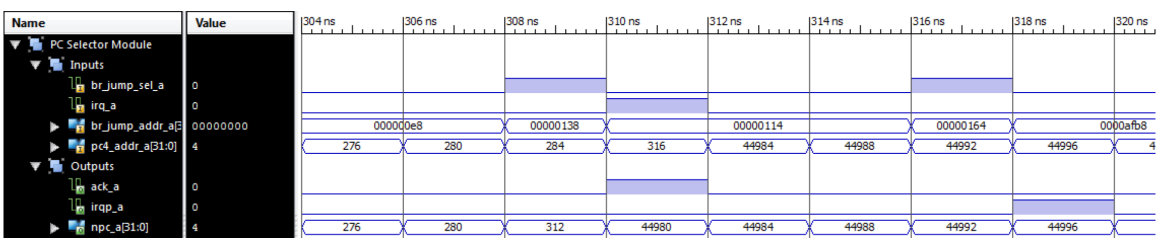

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
224 ns 226 ns 228 ns 230 ns 232 ns 234 ns 236 ns 238 ns 240 ns
00000000
4 8 12 16 20 24 28 32
4 8 12 16 20 24 28 32

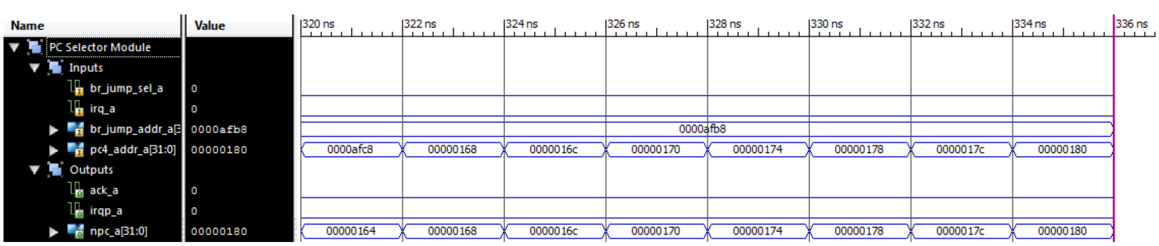

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
240 ns 242 ns 244 ns 246 ns 248 ns 250 ns 252 ns 254 ns 256 ns
00000000
36 40 44 48 52 56 60 64
36 40 44 48 52 56 60 64


Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
256 ns 258 ns 260 ns 262 ns 264 ns 266 ns 268 ns 270 ns 272 ns
00000000
68 72 76 80 84 88 92 96
68 72 76 80 84 88 92 96

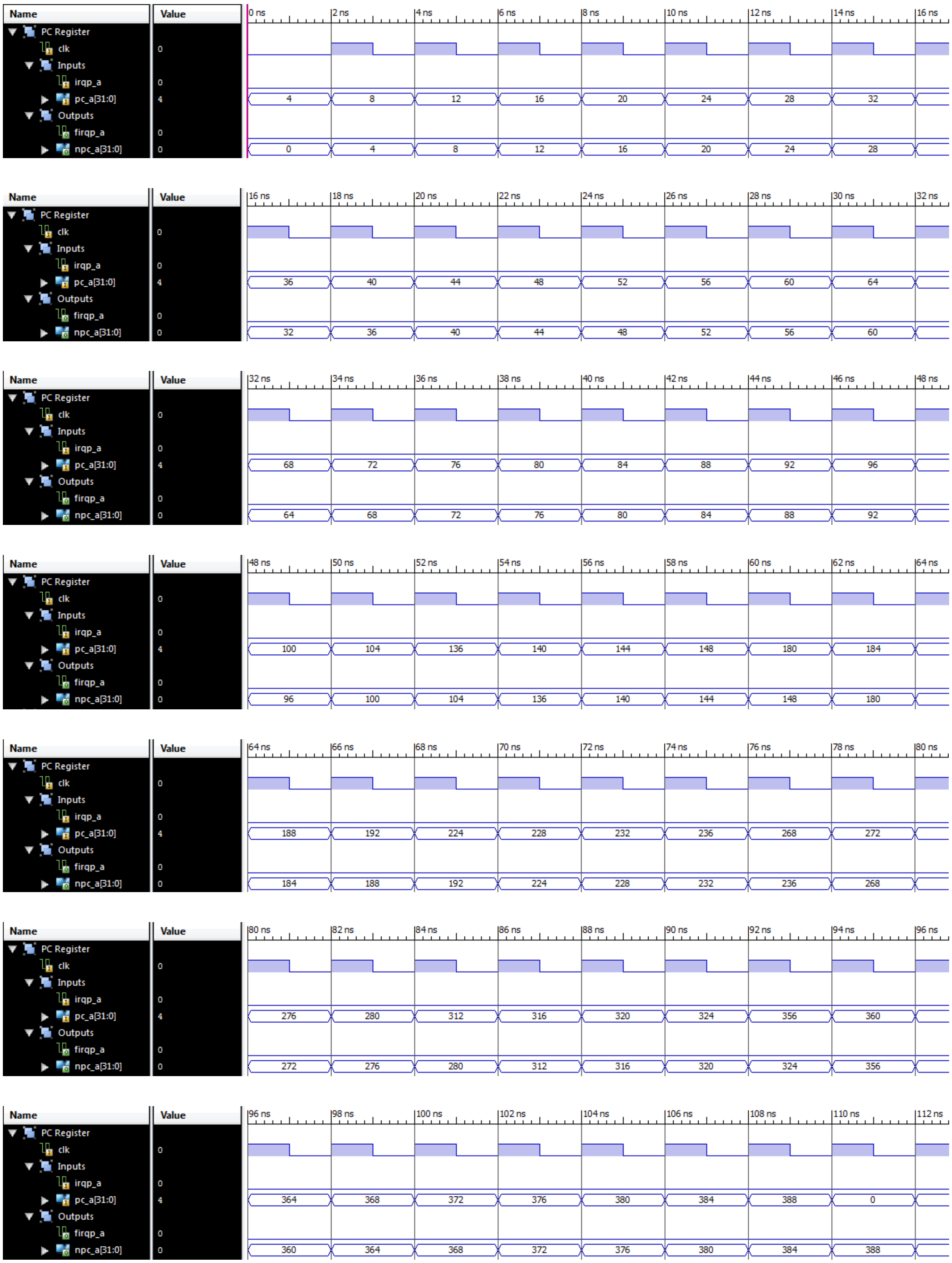

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
272 ns 274 ns 276 ns 278 ns 280 ns 282 ns 284 ns 286 ns 288 ns
00000000 00000088 00000064 000000b4 00000090
100 104 108 140 144 148 152 184
100 104 136 140 144 148 180 184

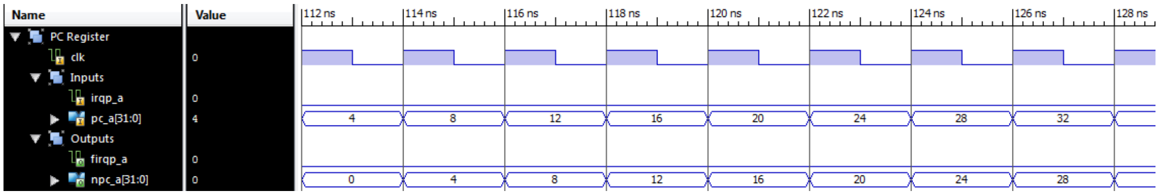

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
288 ns 290 ns 292 ns 294 ns 296 ns 298 ns 300 ns 302 ns 304 ns
00000090 000000e0 000000bc 0000010c 000000e8
188 192 196 228 232 236 240 272
188 192 224 228 232 236 268 272

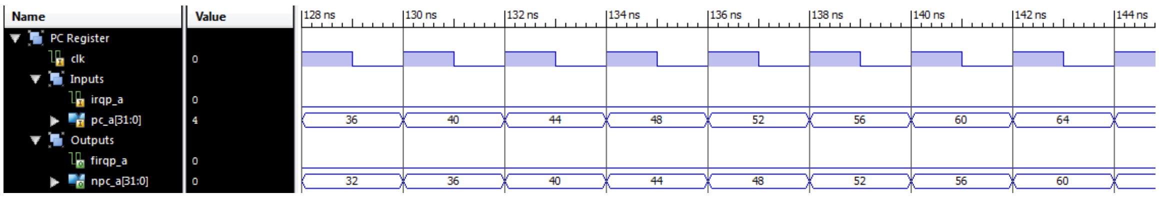

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 00000000
pc4_addr_a[31:0] 4
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 4
304 ns 306 ns 308 ns 310 ns 312 ns 314 ns 316 ns 318 ns 320 ns
000000e8 00000138 00000114 00000164 0000afb8
276 280 284 316 44984 44988 44992 44996
276 280 312 44980 44984 44988 44992 44996

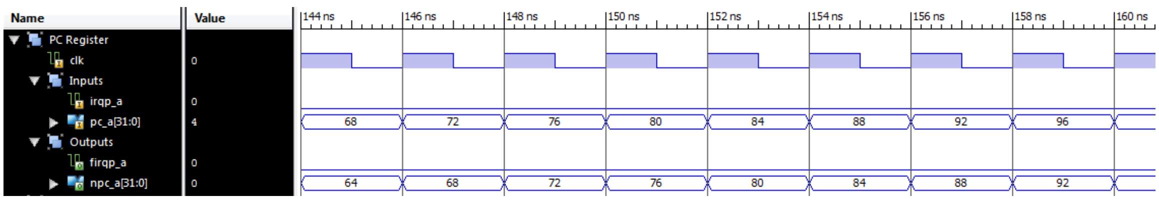

Name
Value
PC Selector Module
Inputs
br_jump_sel_a 0
irq_a 0
br_jump_addr_a[3 0000afb8
pc4_addr_a[31:0] 00000180
Outputs
ack_a 0
irqp_a 0
npc_a[31:0] 00000180
320 ns 322 ns 324 ns 326 ns 328 ns 330 ns 332 ns 334 ns 336 ns
0000afb8
0000afc8 00000168 0000016c 00000170 00000174 00000178 0000017c 00000180
00000164 00000168 0000016c 00000170 00000174 00000178 0000017c 00000180

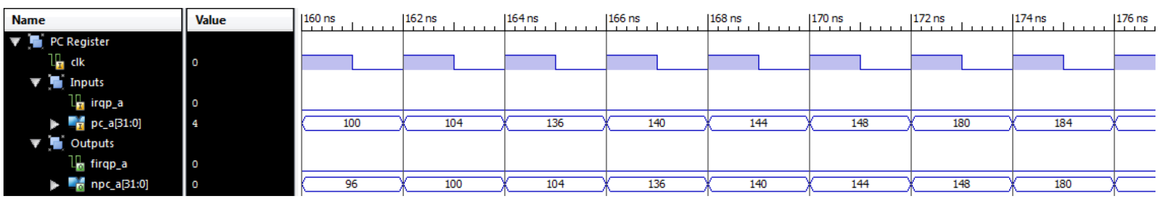
C. PC REGISTER

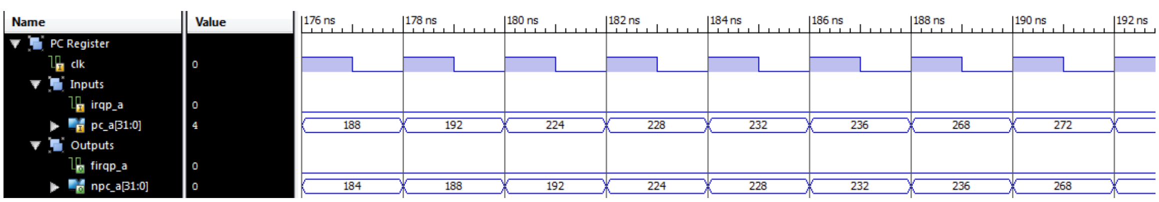


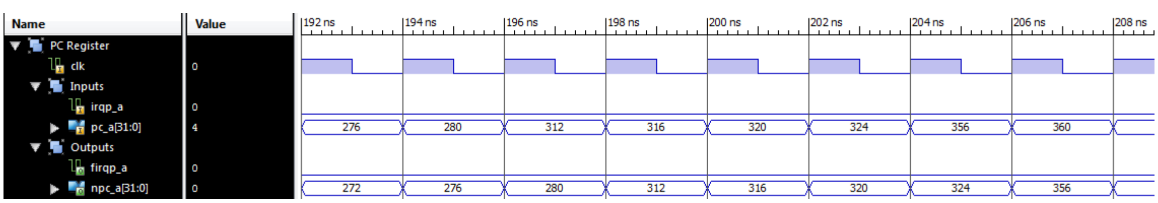

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
112 ns 114 ns 116 ns 118 ns 120 ns 122 ns 124 ns 126 ns 128 ns
4 8 12 16 20 24 28 32
0 4 8 12 16 20 24 28

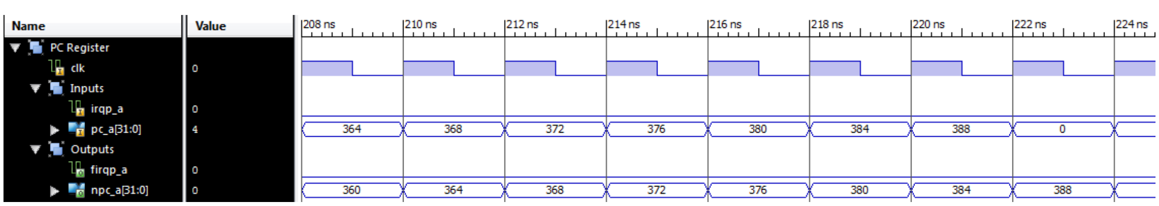

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
128 ns 130 ns 132 ns 134 ns 136 ns 138 ns 140 ns 142 ns 144 ns
36 40 44 48 52 56 60 64
32 36 40 44 48 52 56 60

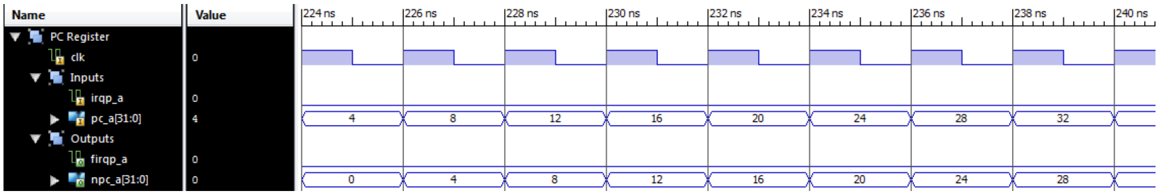

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
144 ns 146 ns 148 ns 150 ns 152 ns 154 ns 156 ns 158 ns 160 ns
68 72 76 80 84 88 92 96
64 68 72 76 80 84 88 92

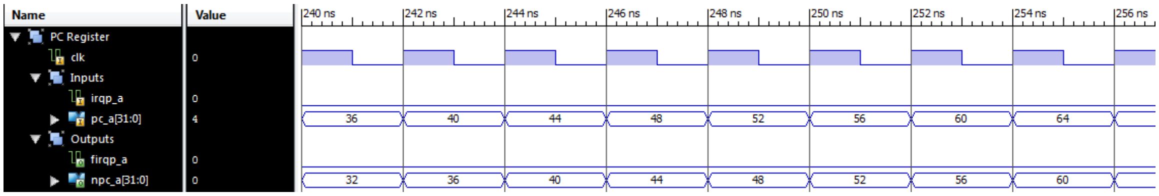

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
160 ns 162 ns 164 ns 166 ns 168 ns 170 ns 172 ns 174 ns 176 ns
100 104 136 140 144 148 180 184
96 100 104 136 140 144 148 180

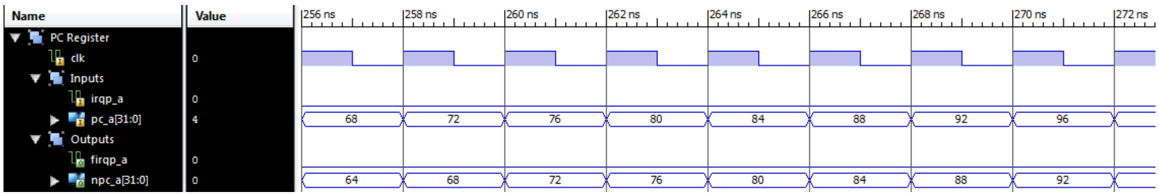

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
176 ns 178 ns 180 ns 182 ns 184 ns 186 ns 188 ns 190 ns 192 ns
188 192 224 228 232 236 268 272
184 188 192 224 228 232 236 268

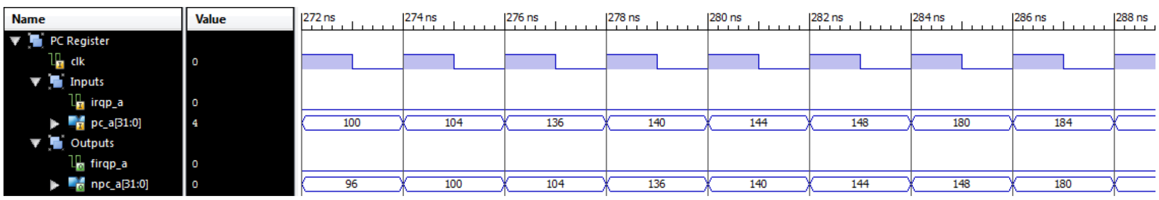

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
192 ns 194 ns 196 ns 198 ns 200 ns 202 ns 204 ns 206 ns 208 ns
276 280 312 316 320 324 356 360
272 276 280 312 316 320 324 356

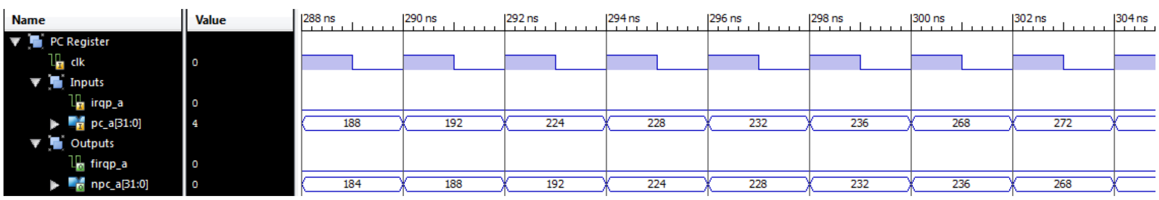

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
208 ns 210 ns 212 ns 214 ns 216 ns 218 ns 220 ns 222 ns 224 ns
364 368 372 376 380 384 388 0
360 364 368 372 376 380 384 388

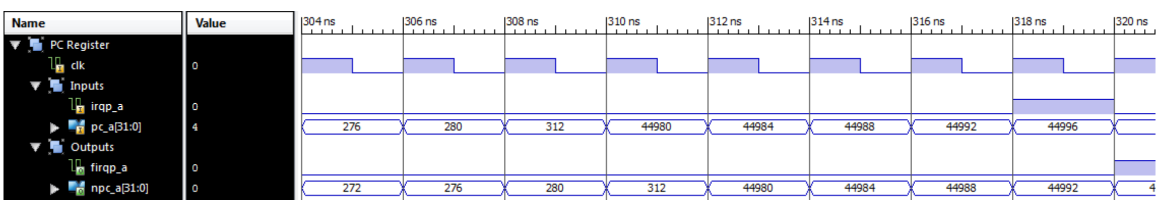

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
224 ns 226 ns 228 ns 230 ns 232 ns 234 ns 236 ns 238 ns 240 ns
4 8 12 16 20 24 28 32
0 4 8 12 16 20 24 28

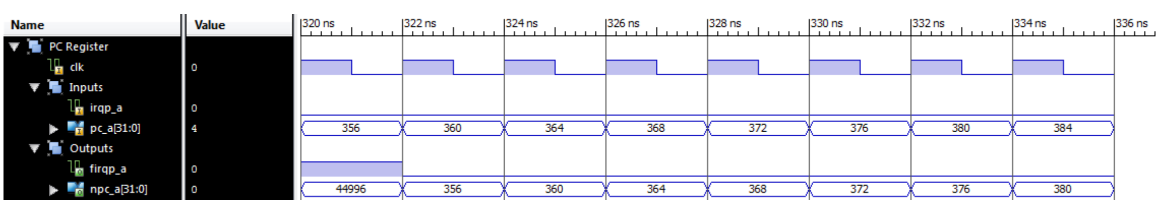

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
240 ns 242 ns 244 ns 246 ns 248 ns 250 ns 252 ns 254 ns 256 ns
36 40 44 48 52 56 60 64
32 36 40 44 48 52 56 60


Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
256 ns 258 ns 260 ns 262 ns 264 ns 266 ns 268 ns 270 ns 272 ns
68 72 76 80 84 88 92 96
64 68 72 76 80 84 88 92

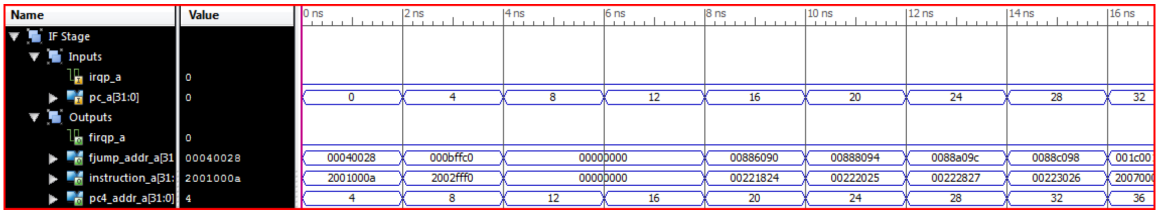

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
272 ns 274 ns 276 ns 278 ns 280 ns 282 ns 284 ns 286 ns 288 ns
100 104 136 140 144 148 180 184
96 100 104 136 140 144 148 180

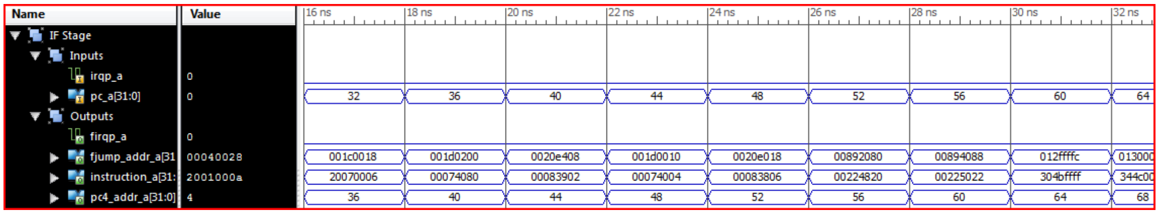

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
288 ns 290 ns 292 ns 294 ns 296 ns 298 ns 300 ns 302 ns 304 ns
188 192 224 228 232 236 268 272
184 188 192 224 228 232 236 268

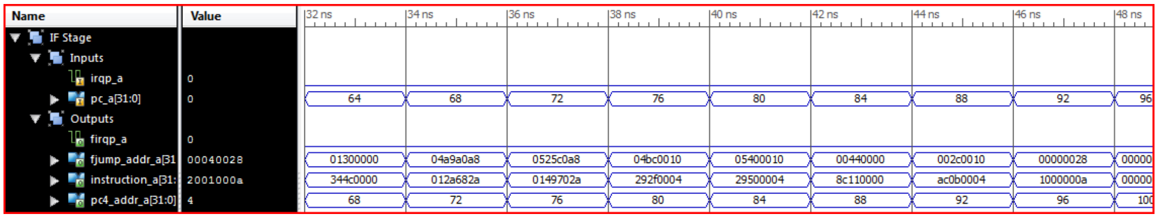

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
304 ns 306 ns 308 ns 310 ns 312 ns 314 ns 316 ns 318 ns 320 ns
276 280 312 44980 44984 44988 44992 44996
272 276 280 312 44980 44984 44988 44992 4

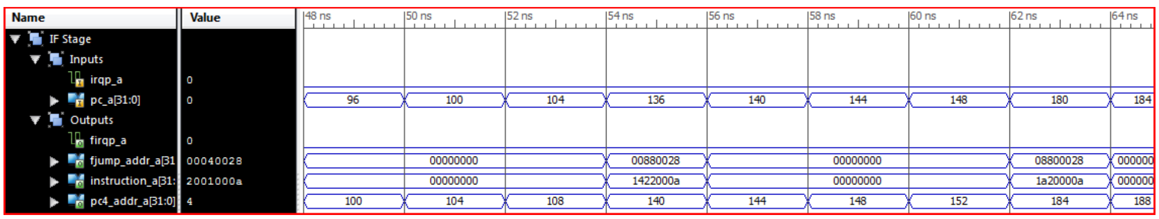

Name
Value
PC Register
clk 0
Inputs
irqp_a 0
pc_a[31:0] 4
Outputs
firqp_a 0
npc_a[31:0] 0
320 ns 322 ns 324 ns 326 ns 328 ns 330 ns 332 ns 334 ns 336 ns
356 360 364 368 372 376 380 384
44996 356 360 364 368 372 376 380

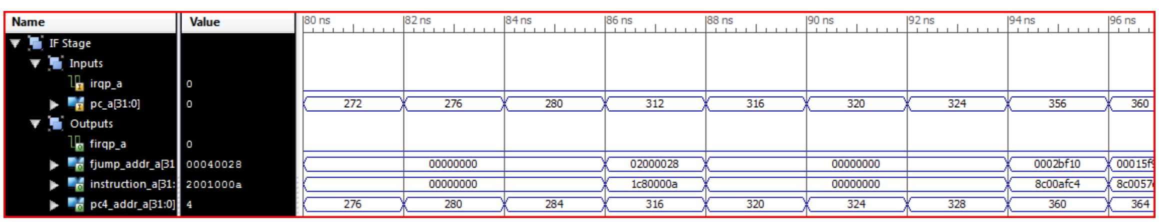
D. IF STAGE

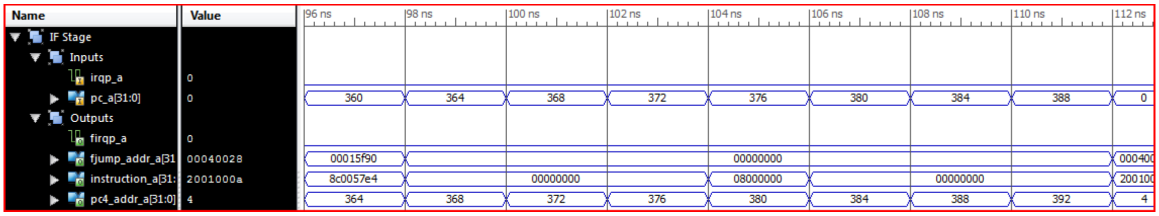


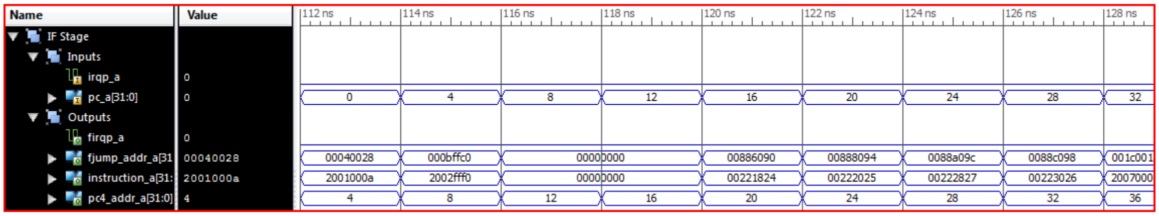


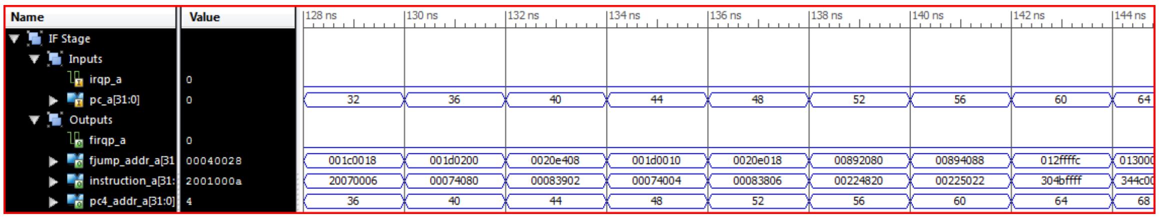


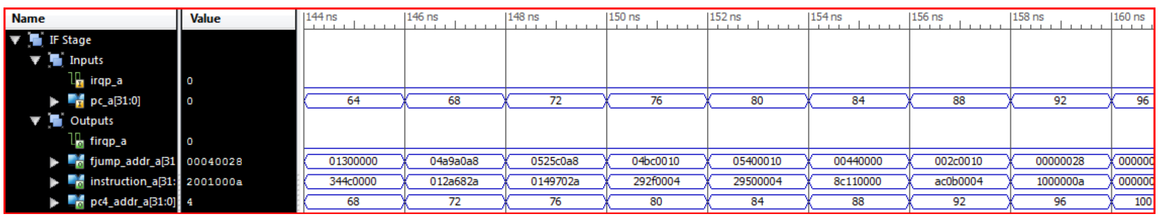


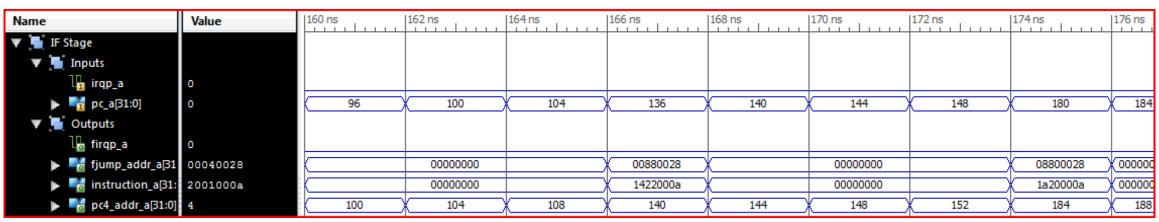


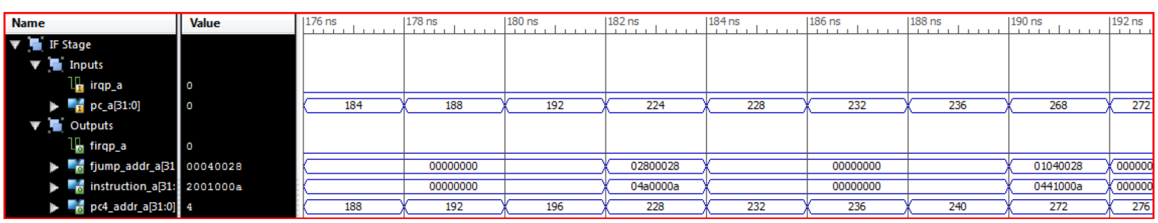

Name
Value
IF Stage
Inputs
irqp_a 0
pc_a[31:0] 0
Outputs
firqp_a 0
fjump_addr_a[31 00040028
instruction_a[31: 2001000a
pc4_addr_a[31:0] 4
96 ns 98 ns 100 ns 102 ns 104 ns 106 ns 108 ns 110 ns 112 ns
360 364 368 372 376 380 384 388 0
00015f90 00000000 00040
8c0057e4 00000000 08000000 00000000 20010
364 368 372 376 380 384 388 392 4

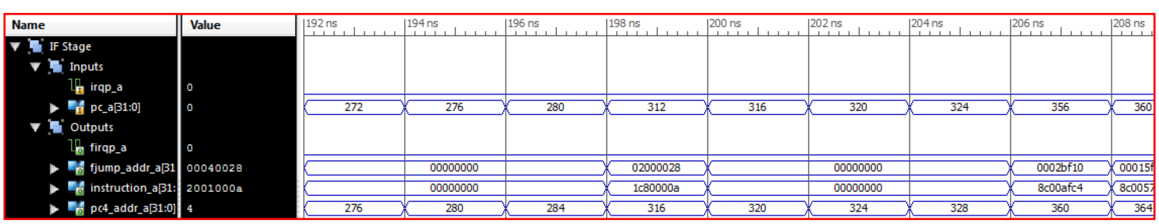

Name
Value
IF Stage
Inputs
irqp_a 0
pc_a[31:0] 0
Outputs
firqp_a 0
fjump_addr_a[31 00040028
instruction_a[31: 2001000a
pc4_addr_a[31:0] 4
112 ns 114 ns 116 ns 118 ns 120 ns 122 ns 124 ns 126 ns 128 ns
0 4 8 12 16 20 24 28 32
00040028 000bffc0 00000000 00886090 00888094 0088a09c 0088c098 001c001
2001000a 2002fff0 00000000 00221824 00222025 00222827 00223026 2007000
4 8 12 16 20 24 28 32 36

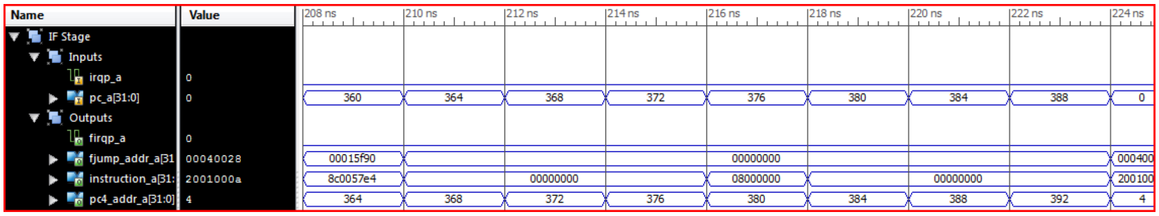

Name
Value
IF Stage
Inputs
irqp_a 0
pc_a[31:0] 0
Outputs
firqp_a 0
fjump_addr_a[31 00040028
instruction_a[31: 2001000a
pc4_addr_a[31:0] 4
128 ns 130 ns 132 ns 134 ns 136 ns 138 ns 140 ns 142 ns 144 ns
32 36 40 44 48 52 56 60 64
001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088 012ffffc 013000
20070006 00074080 00083902 00074004 00083806 00224820 00225022 304bffff 344c00
36 40 44 48 52 56 60 64 68

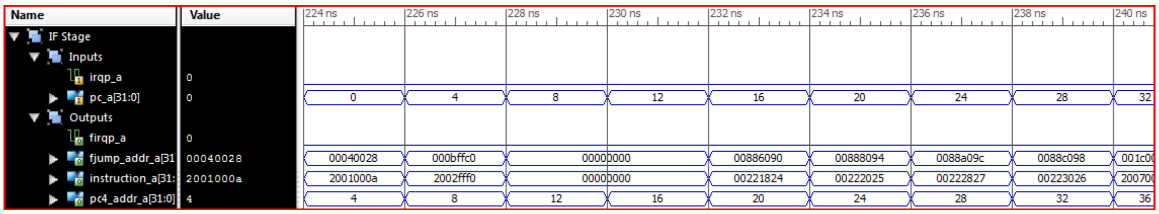

Name
Value
IF Stage
Inputs
irqp_a 0
pc_a[31:0] 0
Outputs
firqp_a 0
fjump_addr_a[31 00040028
instruction_a[31: 2001000a
pc4_addr_a[31:0] 4
144 ns 146 ns 148 ns 150 ns 152 ns 154 ns 156 ns 158 ns 160 ns
64 68 72 76 80 84 88 92 96
01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010 00000028 00000
344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004 1000000a 00000
68 72 76 80 84 88 92 96 100

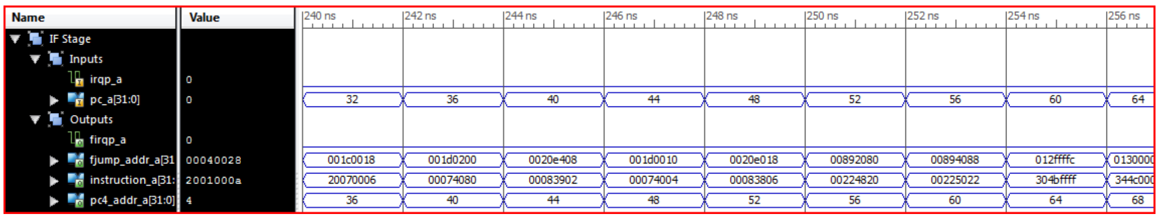

Name
Value
IF Stage
Inputs
irqp_a 0
pc_a[31:0] 0
Outputs
firqp_a 0
fjump_addr_a[31 00040028
instruction_a[31: 2001000a
pc4_addr_a[31:0] 4
160 ns 162 ns 164 ns 166 ns 168 ns 170 ns 172 ns 174 ns 176 ns
96 100 104 136 140 144 148 180 184
00000000 00880028 00000000 08800028 00000
00000000 1422000a 00000000 1a20000a 00000
100 104 108 140 144 148 152 184 188

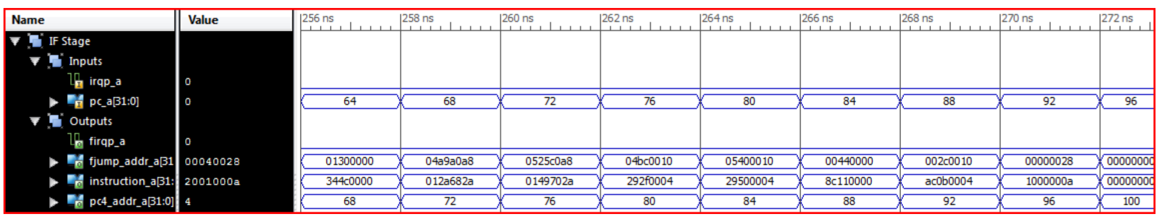

Name
Value
IF Stage
Inputs
irqp_a 0
pc_a[31:0] 0
Outputs
firqp_a 0
fjump_addr_a[31 00040028
instruction_a[31: 2001000a
pc4_addr_a[31:0] 4
176 ns 178 ns 180 ns 182 ns 184 ns 186 ns 188 ns 190 ns 192 ns
184 188 192 224 228 232 236 268 272
00000000 02800028 00000000 01040028 00000
00000000 04a0000a 00000000 0441000a 00000
188 192 196 228 232 236 240 272 276

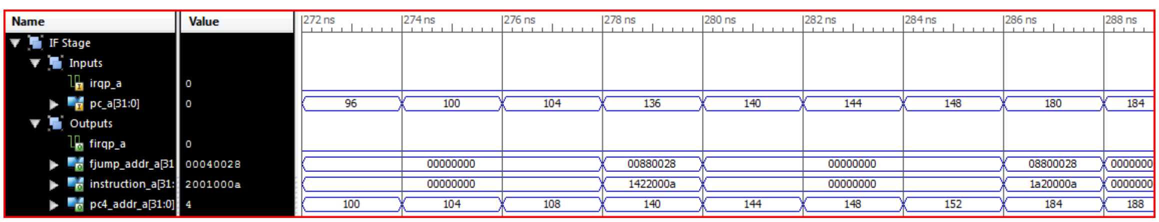

Name
Value
IF Stage
Inputs
irqp_a 0
pc_a[31:0] 0
Outputs
firqp_a 0
fjump_addr_a[31 00040028
instruction_a[31: 2001000a
pc4_addr_a[31:0] 4
192 ns 194 ns 196 ns 198 ns 200 ns 202 ns 204 ns 206 ns 208 ns
272 276 280 312 316 320 324 356 360
00000000 02000028 00000000 0002bf10 00015f
00000000 1c80000a 00000000 8c00afc4 8c0057
276 280 284 316 320 324 328 360 364

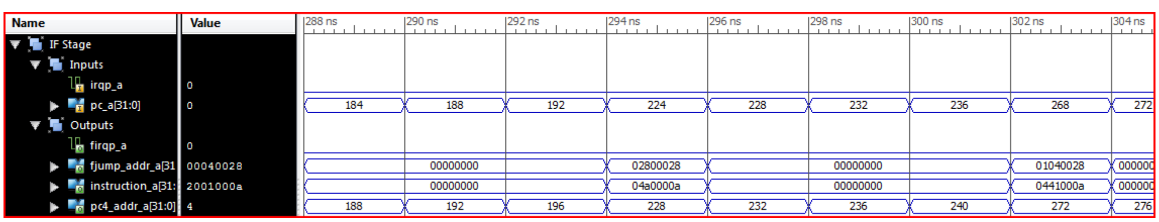


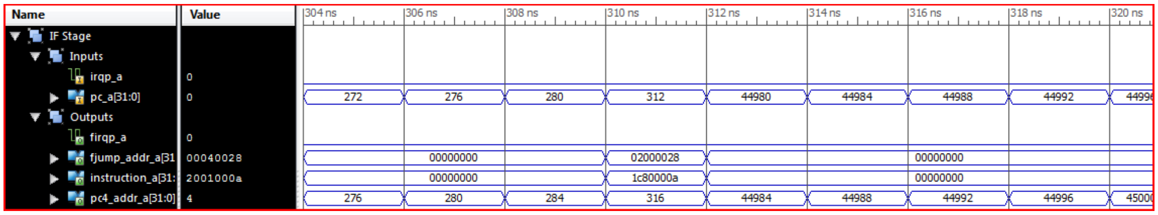


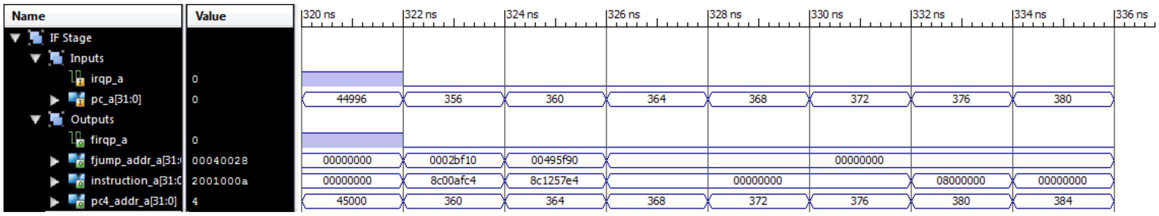




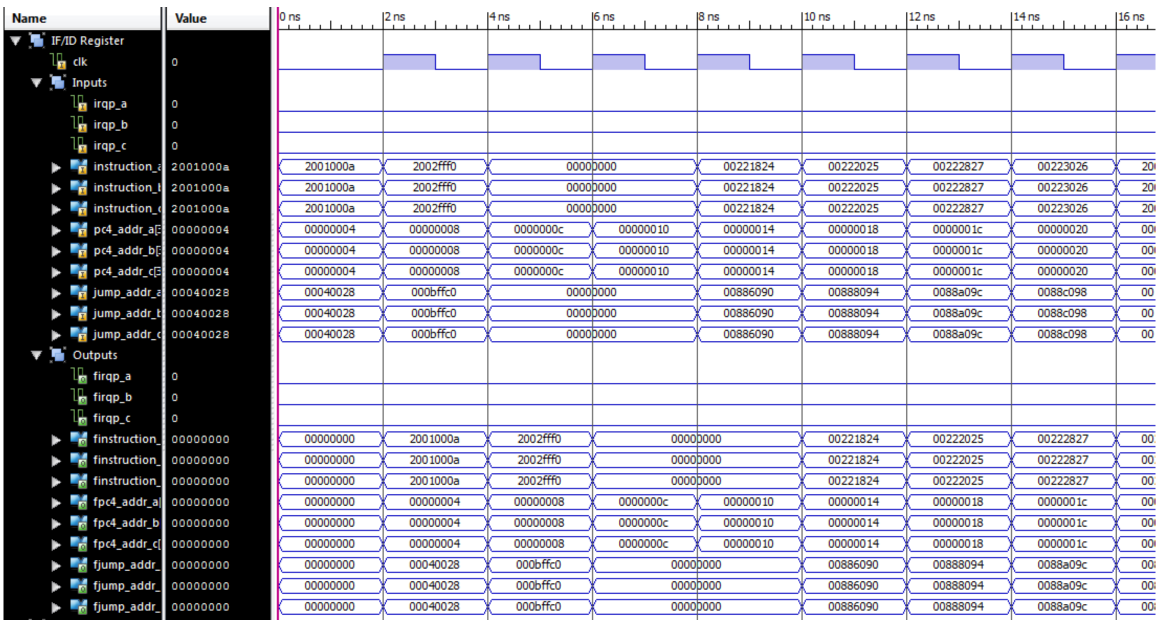


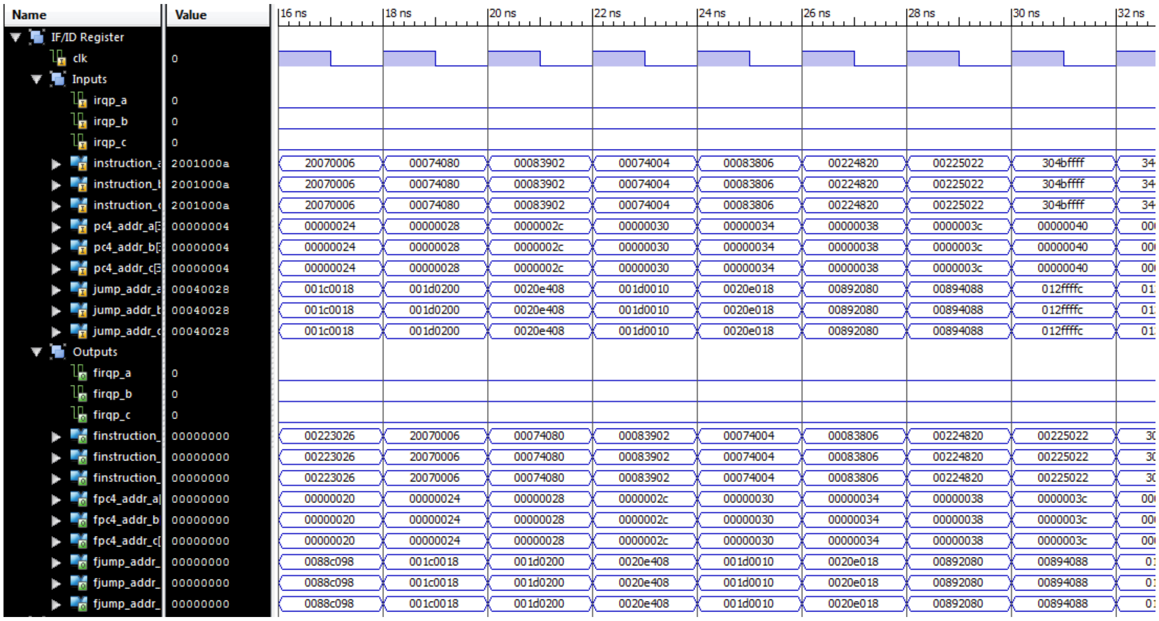


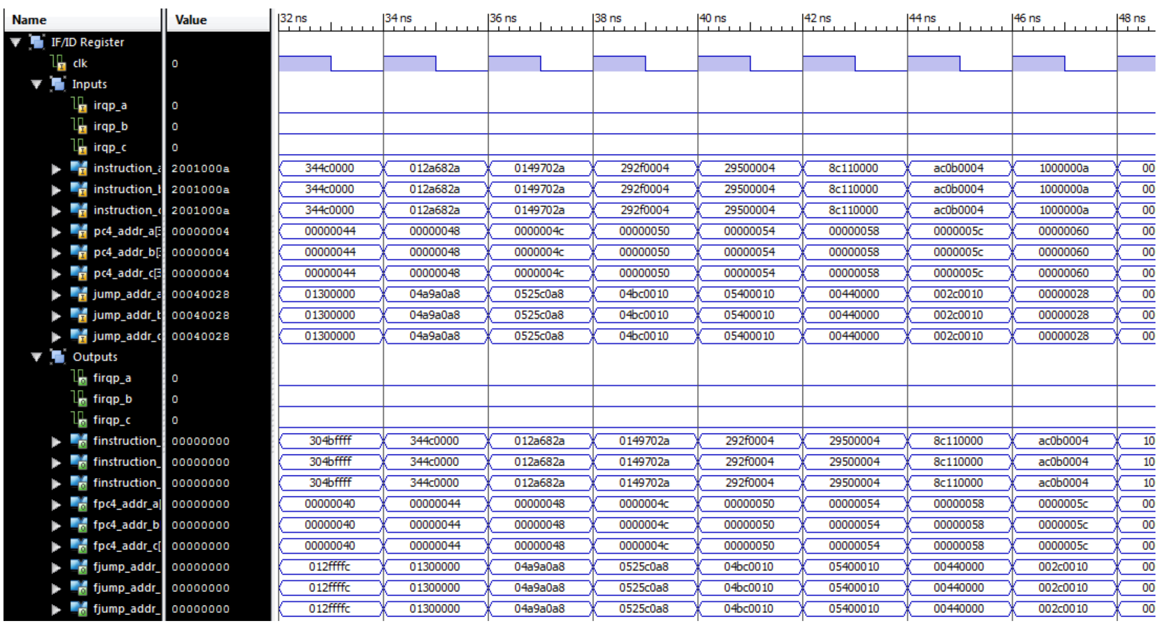


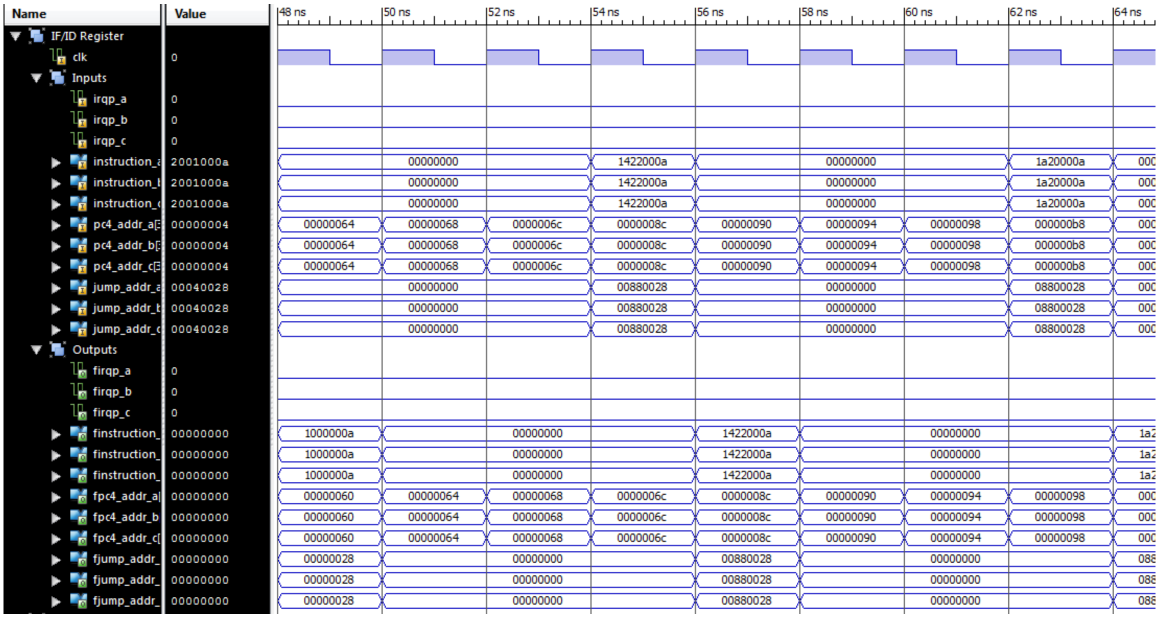


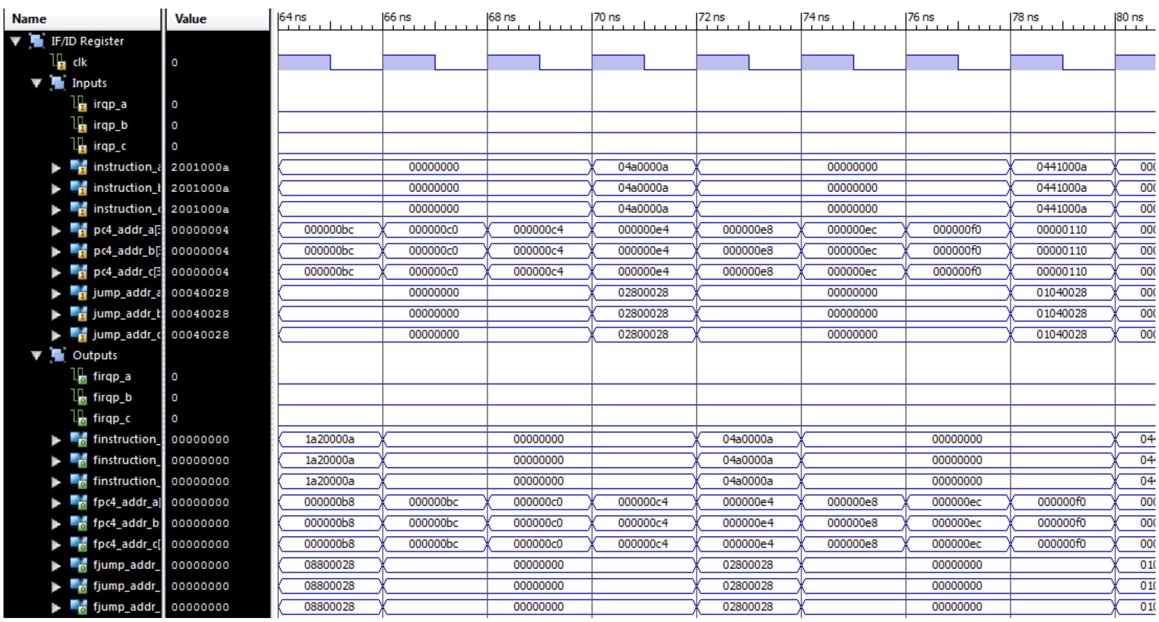
E. IF/ID REGISTER

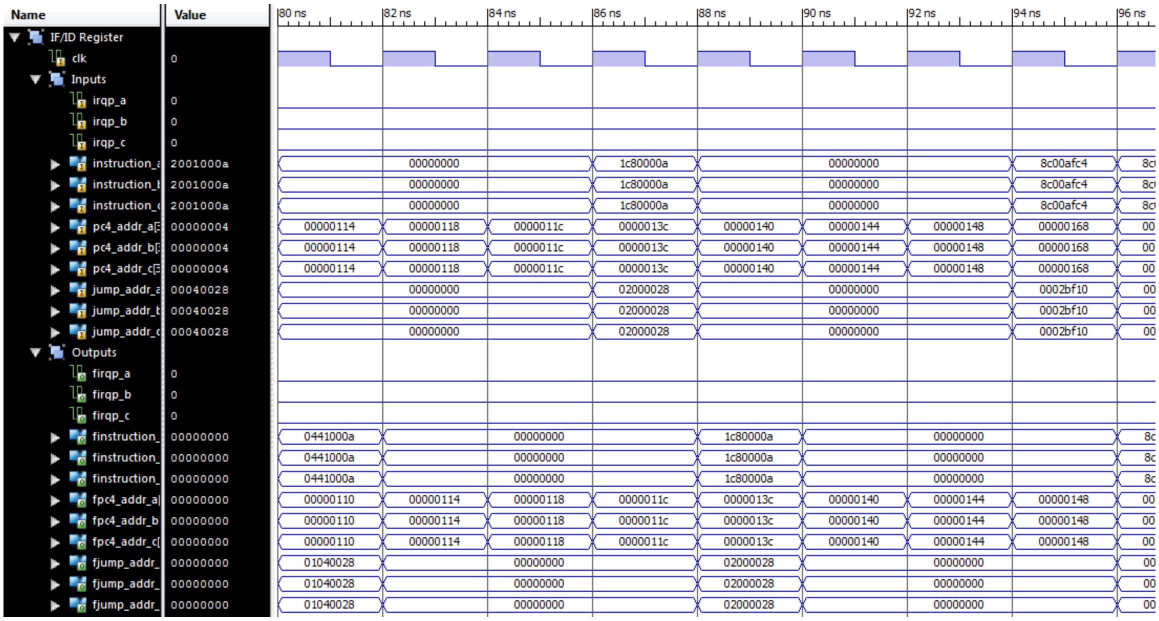


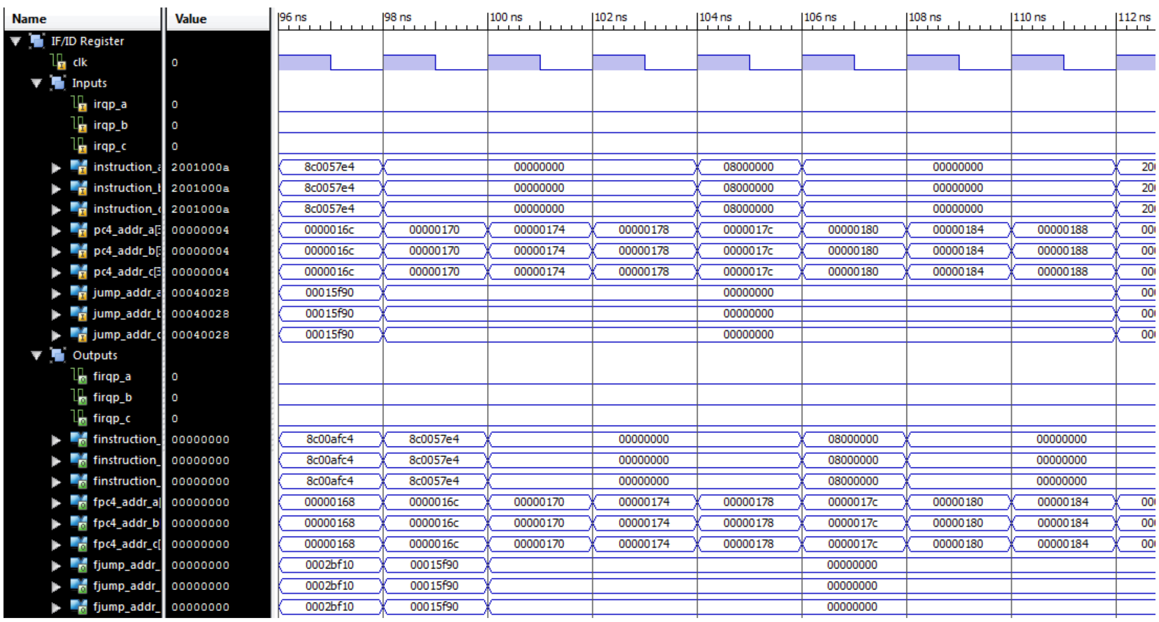

Name
Value
16 ns
18 ns
20 ns
22 ns
24 ns
26 ns
28 ns
30 ns
32 ns
IF/ID Register
clk 0
Inputs
irqp_a 0
irqp_b 0
irqp_c 0
instruction_ 2001000a 20070006 00074080 00083902 00074004 00083806 00224820 00225022 304bffff
instruction_ 2001000a 20070006 00074080 00083902 00074004 00083806 00224820 00225022 304bffff
instruction_ 2001000a 20070006 00074080 00083902 00074004 00083806 00224820 00225022 304bffff
pc4_addr_a 00000004 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040
pc4_addr_b 00000004 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040
pc4_addr_c 00000004 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040
jump_addr_ 00040028 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088 012ffffc
jump_addr_ 00040028 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088 012ffffc
jump_addr_ 00040028 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088 012ffffc
Outputs
firqp_a 0
firqp_b 0
firqp_c 0
finstruction_ 00000000 00223026 20070006 00074080 00083902 00074004 00083806 00224820 00225022
finstruction_ 00000000 00223026 20070006 00074080 00083902 00074004 00083806 00224820 00225022
finstruction_ 00000000 00223026 20070006 00074080 00083902 00074004 00083806 00224820 00225022
fpc4_addr_a 00000000 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c
fpc4_addr_b 00000000 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c
fpc4_addr_c 00000000 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c
fjump_addr_ 00000000 0088c098 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088
fjump_addr_ 00000000 0088c098 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088
fjump_addr_ 00000000 0088c098 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088

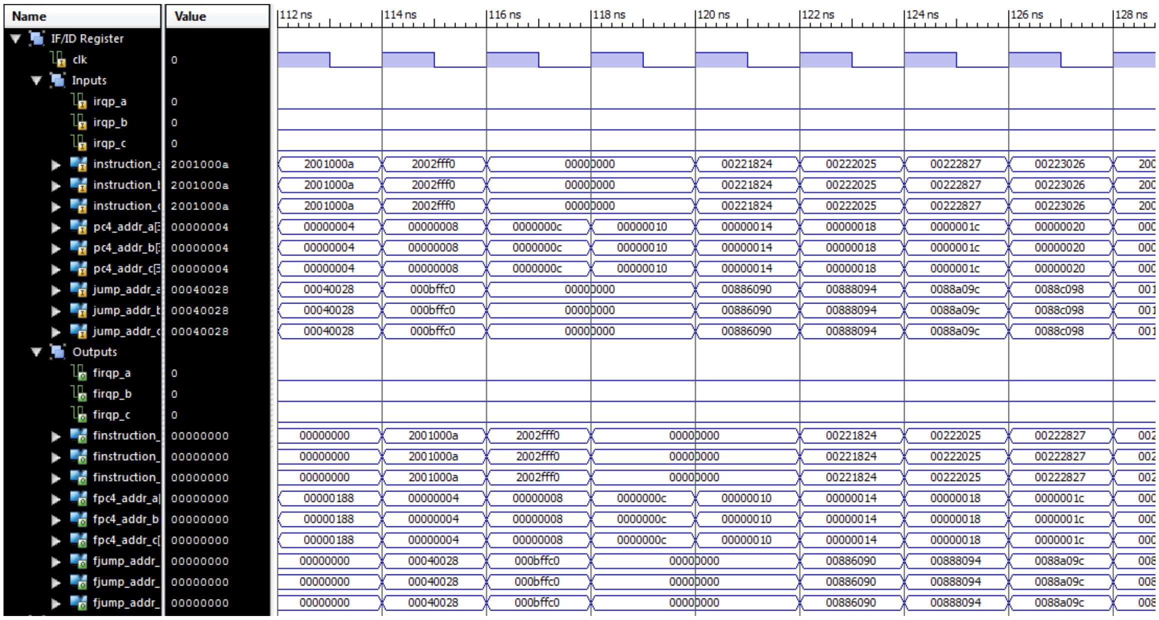

Name
Value
32 ns
34 ns
36 ns
38 ns
40 ns
42 ns
44 ns
46 ns
48 ns
IF/ID Register
clk 0
Inputs
irqp_a 0
irqp_b 0
irqp_c 0
instruction_ 2001000a 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004 1000000a
instruction_ 2001000a 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004 1000000a
instruction_ 2001000a 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004 1000000a
pc4_addr_a 00000004 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c 00000060
pc4_addr_b 00000004 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c 00000060
pc4_addr_c 00000004 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c 00000060
jump_addr_ 00040028 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010 00000028
jump_addr_ 00040028 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010 00000028
jump_addr_ 00040028 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010 00000028
Outputs
firqp_a 0
firqp_b 0
firqp_c 0
finstruction_ 00000000 304bffff 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004
finstruction_ 00000000 304bffff 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004
finstruction_ 00000000 304bffff 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004
fpc4_addr_a 00000000 00000040 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c
fpc4_addr_b 00000000 00000040 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c
fpc4_addr_c 00000000 00000040 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c
fjump_addr_ 00000000 012ffffc 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010
fjump_addr_ 00000000 012ffffc 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010
fjump_addr_ 00000000 012ffffc 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010

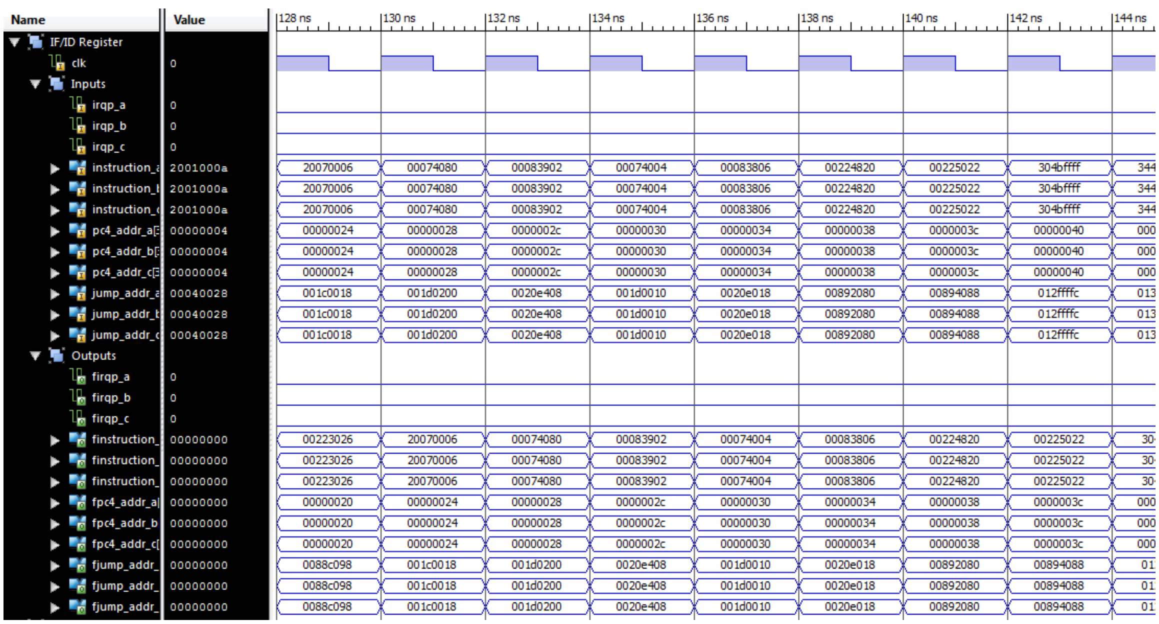

Name
Value
48 ns
50 ns
52 ns
54 ns
56 ns
58 ns
60 ns
62 ns
64 ns
IF/ID Register
clk 0
Inputs
irqp_a 0
irqp_b 0
irqp_c 0
instruction_ 2001000a
instruction_ 2001000a
instruction_ 2001000a
pc4_addr_a 00000004
pc4_addr_b 00000004
pc4_addr_c 00000004
jump_addr_ 00040028
jump_addr_ 00040028
jump_addr_ 00040028
Outputs
firqp_a 0
firqp_b 0
firqp_c 0
finstruction_ 00000000
finstruction_ 00000000
finstruction_ 00000000
fpc4_addr_a 00000000
fpc4_addr_b 00000000
fpc4_addr_c 00000000
fjump_addr_ 00000000
fjump_addr_ 00000000
fjump_addr_ 00000000

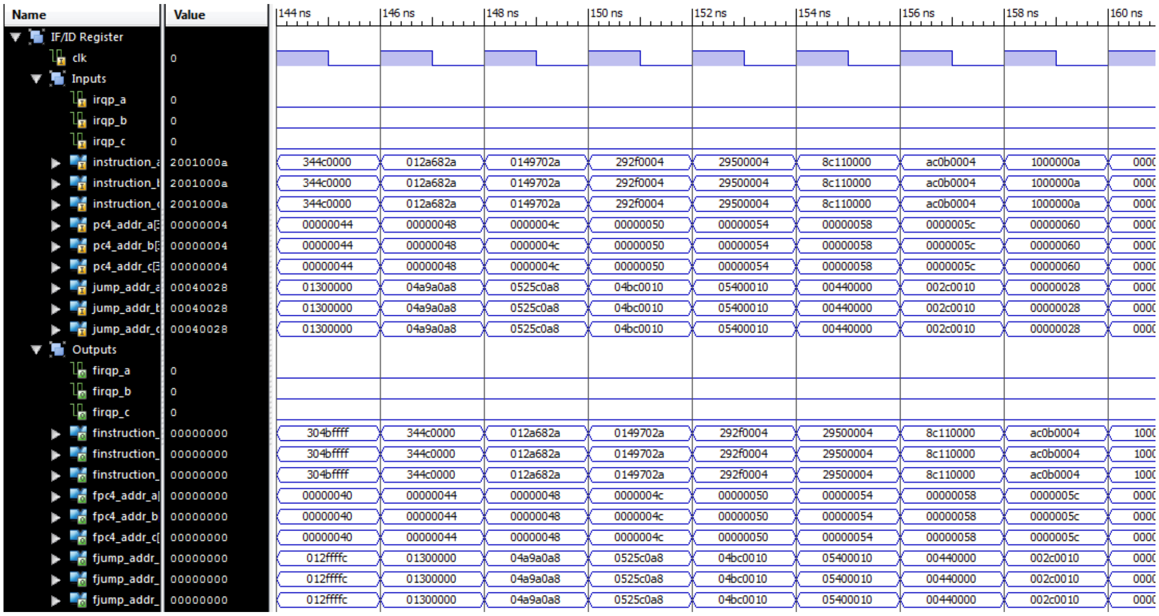

Name
Value
64 ns
66 ns
68 ns
70 ns
72 ns
74 ns
76 ns
78 ns
80 ns
IF/ID Register
clk 0
Inputs
irqp_a 0
irqp_b 0
irqp_c 0
instruction_ 2001000a
instruction_ 2001000a
instruction_ 2001000a
pc4_addr_a 00000004
pc4_addr_b 00000004
pc4_addr_c 00000004
jump_addr_ 00040028
jump_addr_ 00040028
jump_addr_ 00040028
Outputs
firqp_a 0
firqp_b 0
firqp_c 0
finstruction_ 00000000
finstruction_ 00000000
finstruction_ 00000000
fpc4_addr_a 00000000
fpc4_addr_b 00000000
fpc4_addr_c 00000000
fjump_addr_ 00000000
fjump_addr_ 00000000
fjump_addr_ 00000000

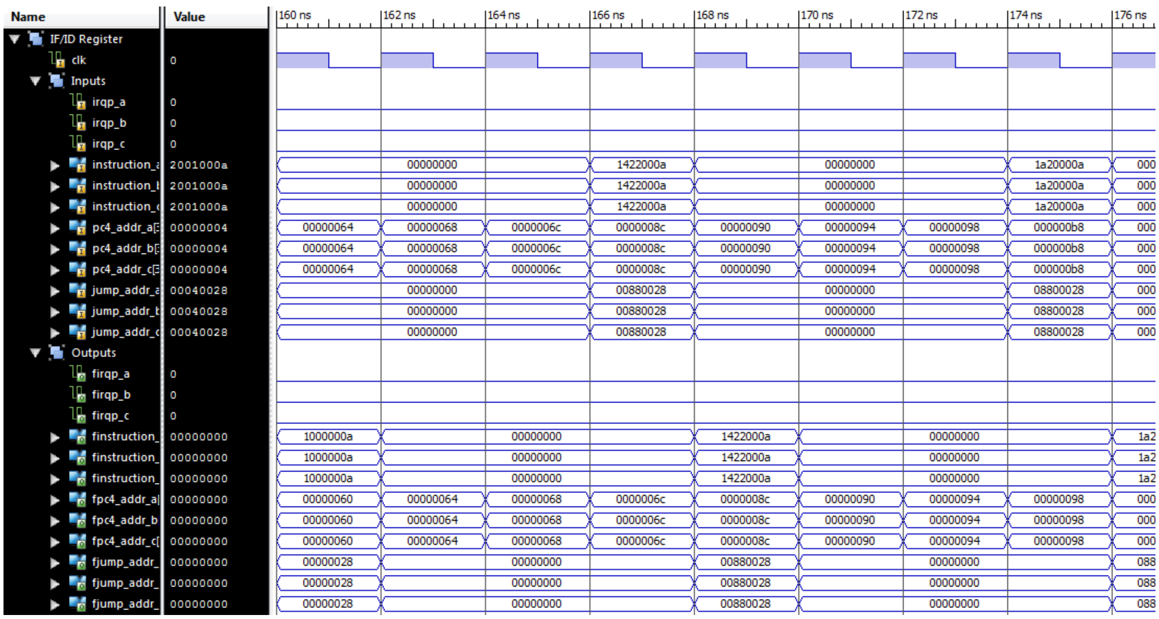


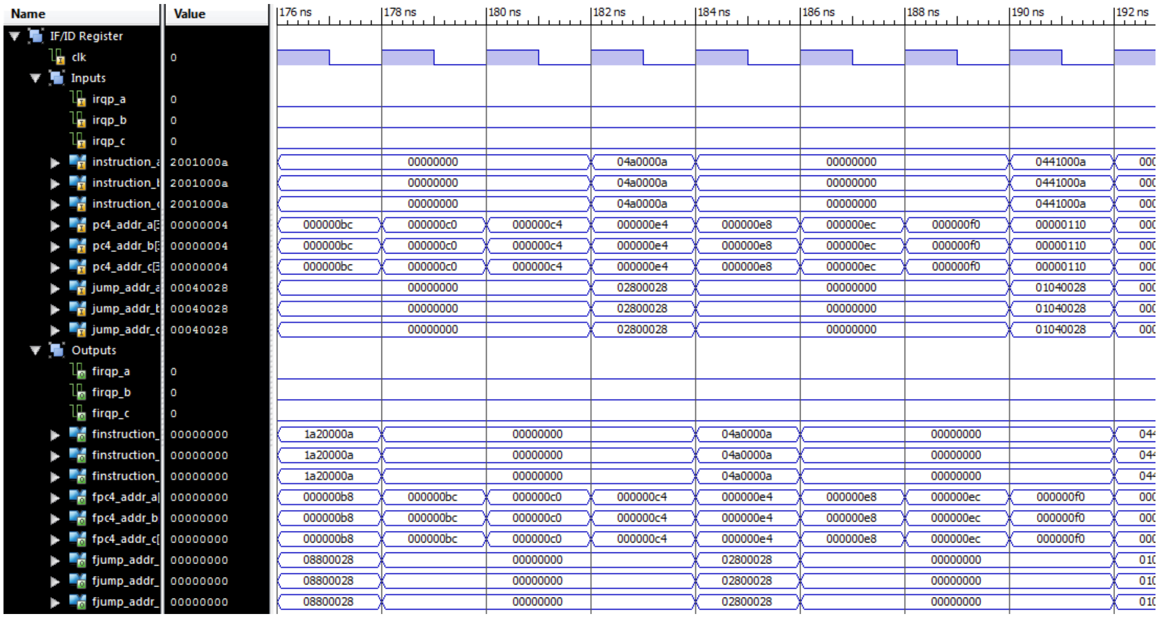


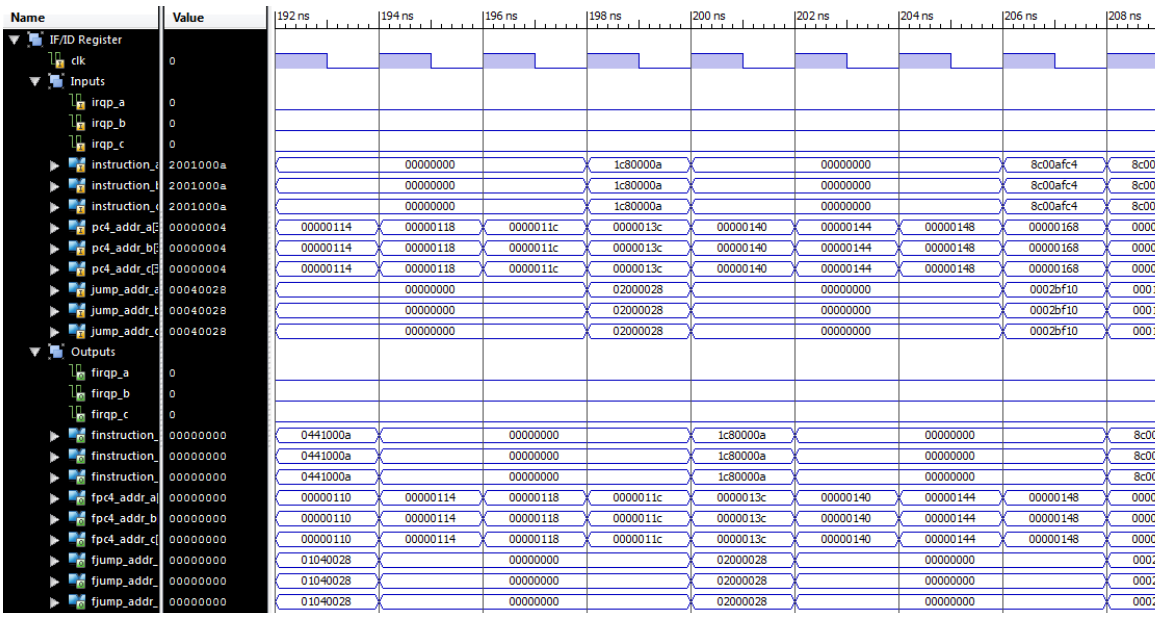


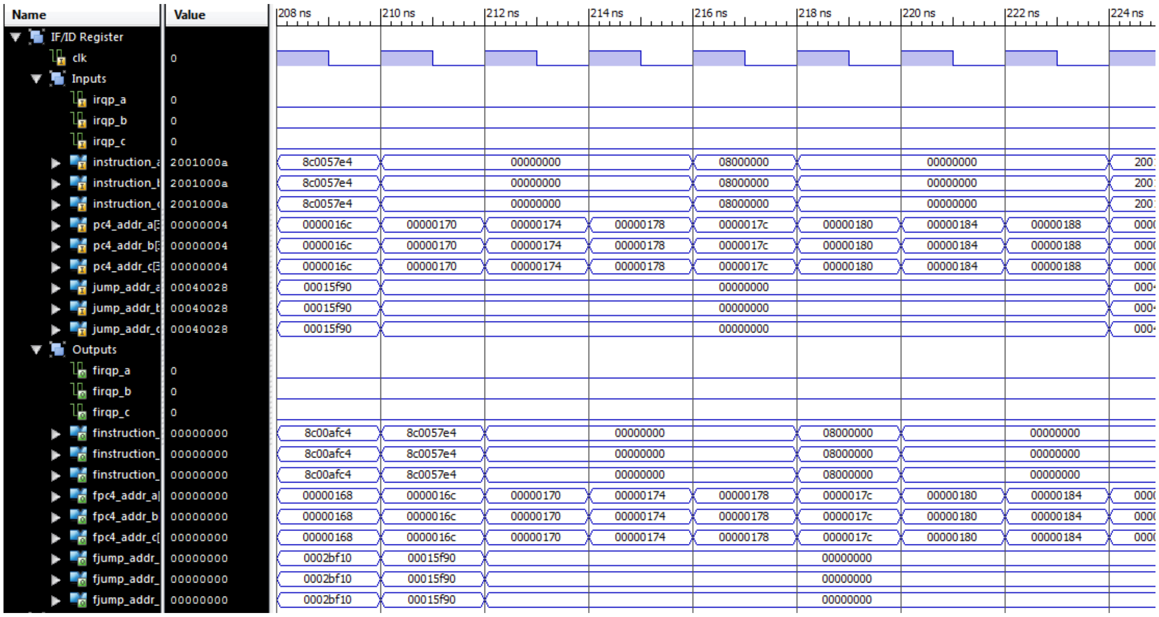


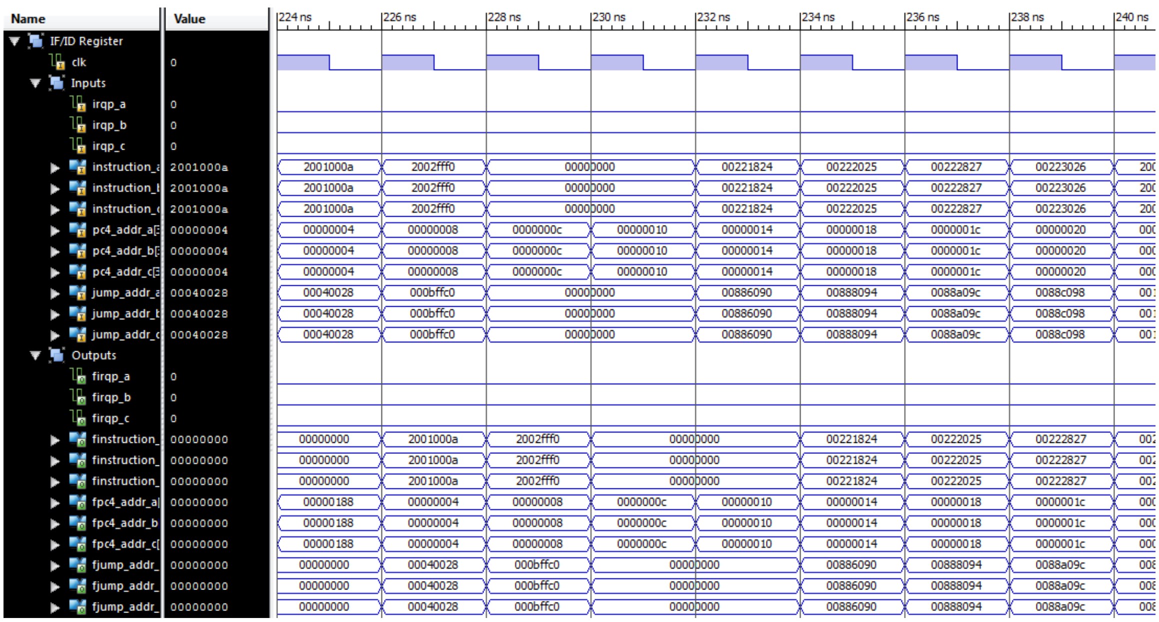


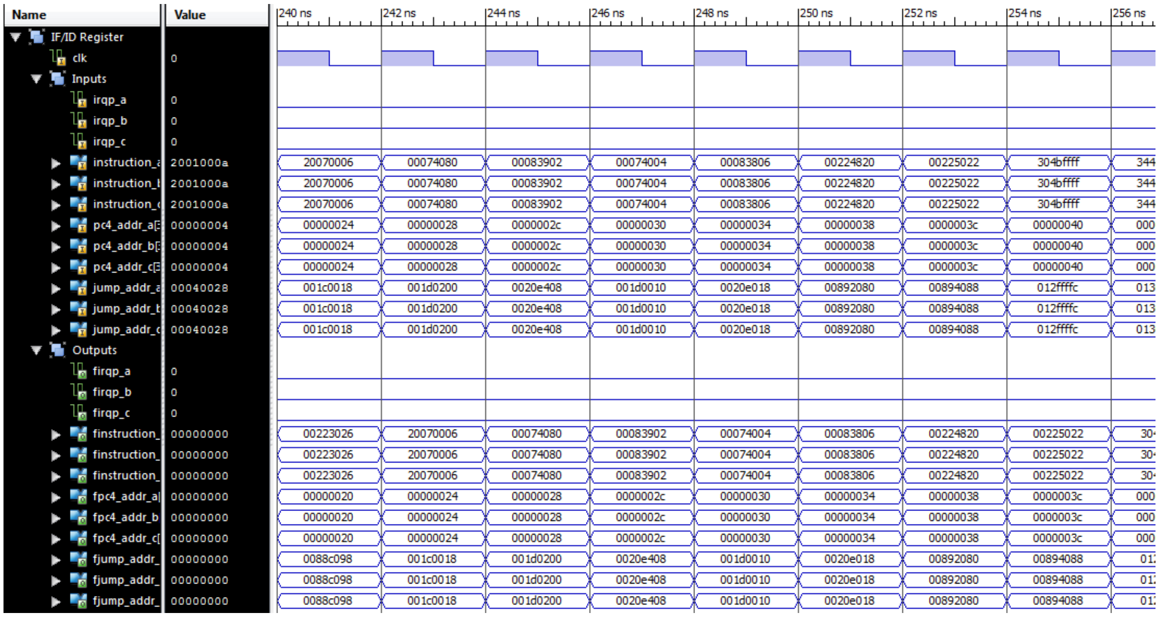




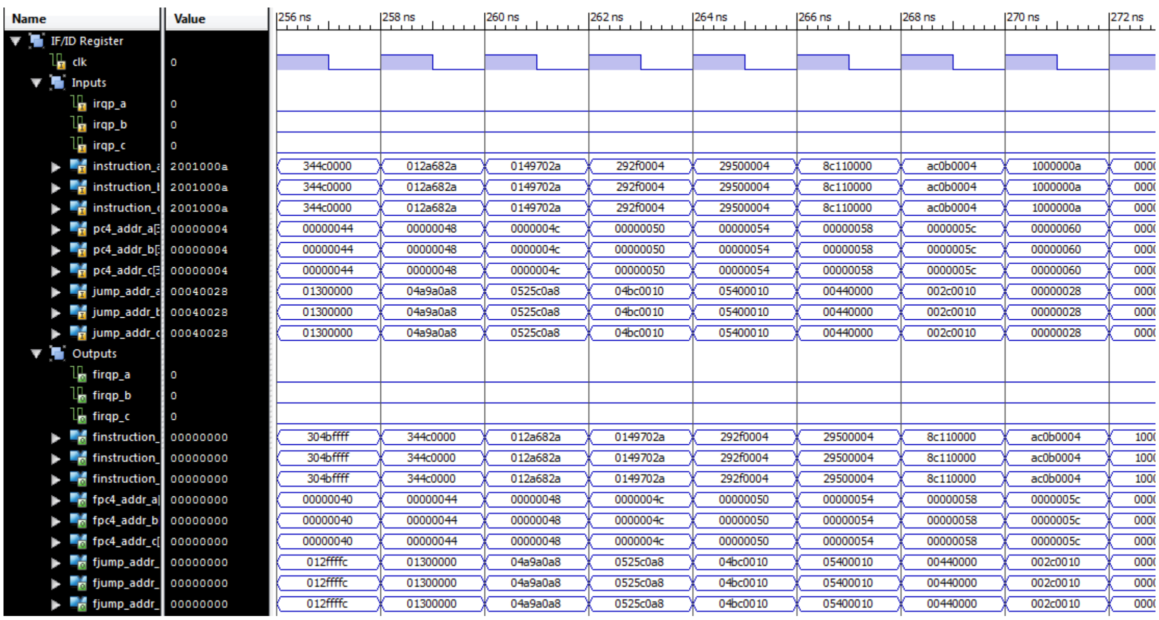




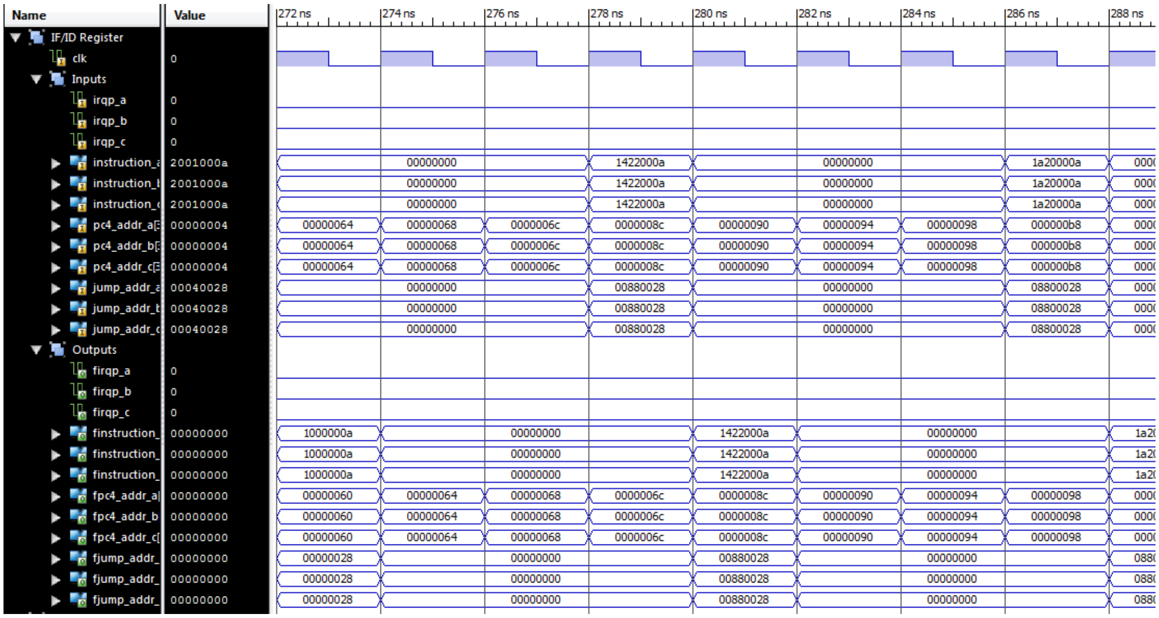


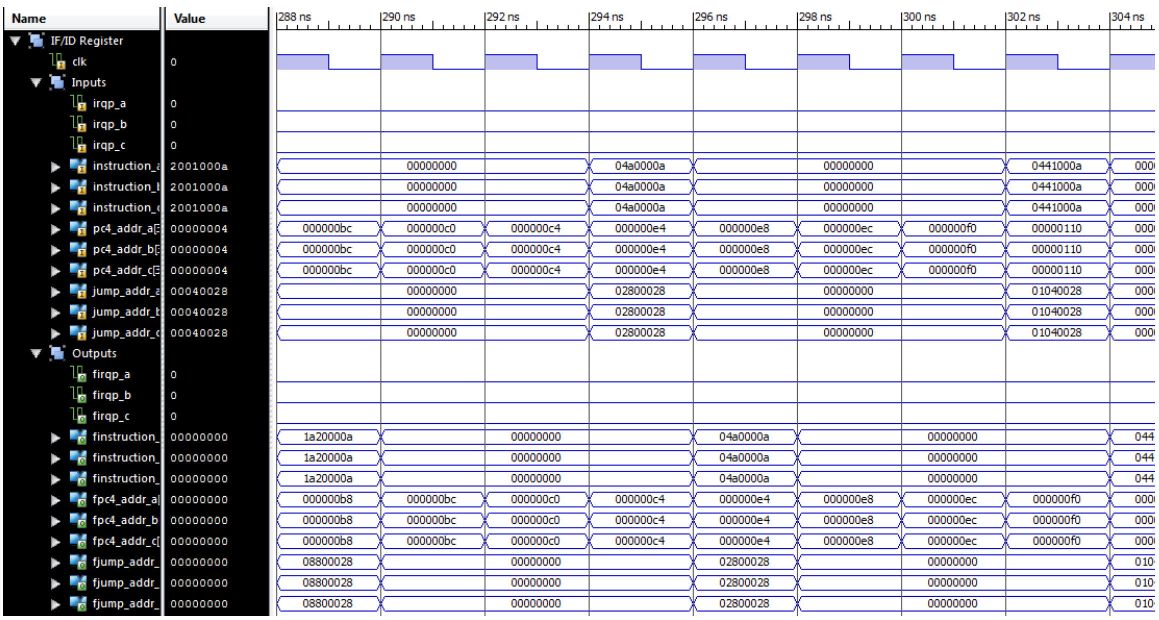


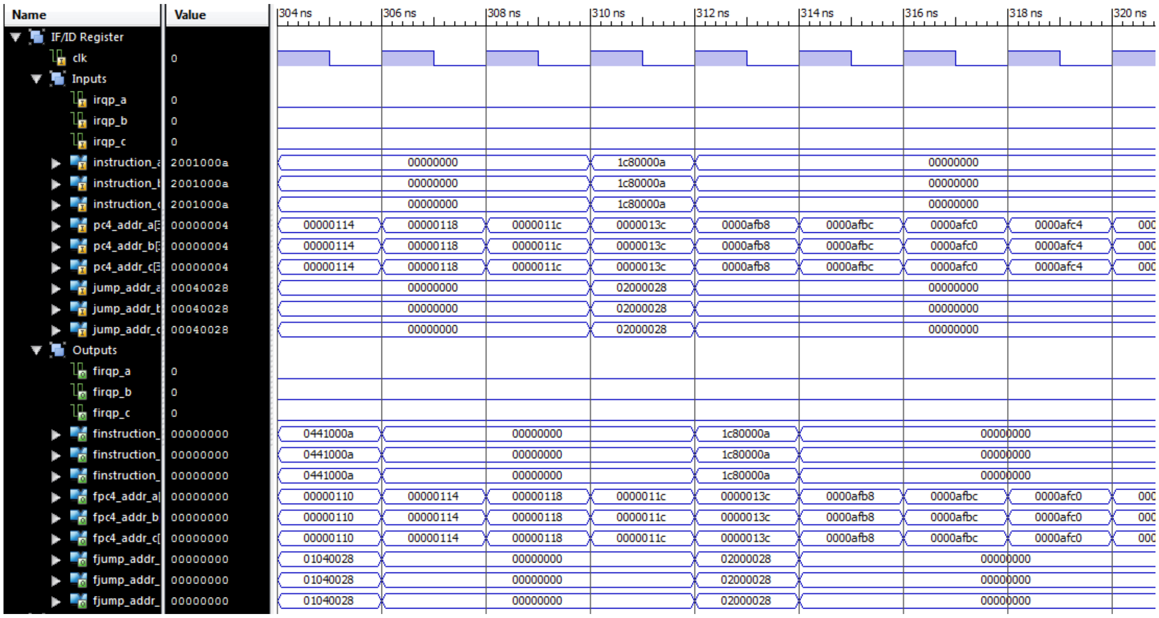
Name
Value
240 ns
242 ns
244 ns
246 ns
248 ns
250 ns
252 ns
254 ns
256 ns
IF/ID Register
clk 0
Inputs
irqp_a 0
irqp_b 0
irqp_c 0
instruction_a 2001000a 20070006 00074080 00083902 00074004 00083806 00224820 00225022 304bffff
instruction_b 2001000a 20070006 00074080 00083902 00074004 00083806 00224820 00225022 304bffff
instruction_c 2001000a 20070006 00074080 00083902 00074004 00083806 00224820 00225022 304bffff
pc4_addr_a 00000004 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040
pc4_addr_b 00000004 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040
pc4_addr_c 00000004 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c 00000040
jump_addr_a 00040028 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088 012ffffc
jump_addr_b 00040028 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088 012ffffc
jump_addr_c 00040028 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088 012ffffc
Outputs
firqp_a 0
firqp_b 0
firqp_c 0
finstruction_ 00000000 00223026 20070006 00074080 00083902 00074004 00083806 00224820 00225022
finstruction_ 00000000 00223026 20070006 00074080 00083902 00074004 00083806 00224820 00225022
finstruction_ 00000000 00223026 20070006 00074080 00083902 00074004 00083806 00224820 00225022
fpc4_addr_a 00000000 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c
fpc4_addr_b 00000000 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c
fpc4_addr_c 00000000 00000020 00000024 00000028 0000002c 00000030 00000034 00000038 0000003c
fjump_addr_ 00000000 0088c098 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088
fjump_addr_ 00000000 0088c098 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088
fjump_addr_ 00000000 0088c098 001c0018 001d0200 0020e408 001d0010 0020e018 00892080 00894088

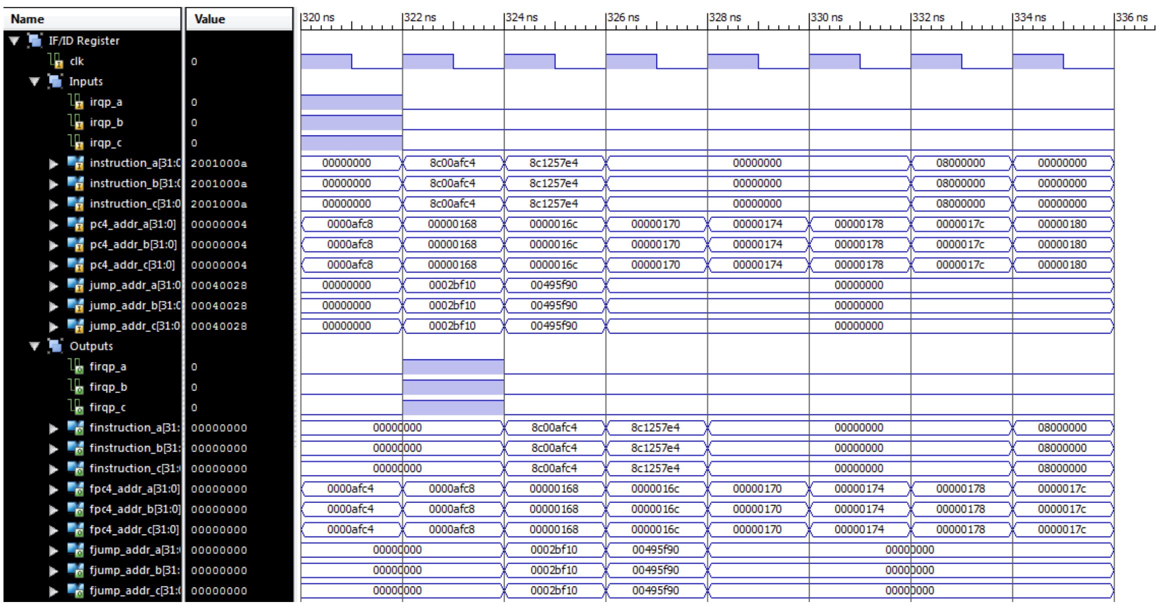


Name
Value
256 ns
258 ns
260 ns
262 ns
264 ns
266 ns
268 ns
270 ns
272 ns
IF/ID Register
clk 0
Inputs
irqp_a 0
irqp_b 0
irqp_c 0
instruction_a 2001000a 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004 1000000a
instruction_b 2001000a 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004 1000000a
instruction_c 2001000a 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004 1000000a
pc4_addr_a 00000004 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c 00000060
pc4_addr_b 00000004 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c 00000060
pc4_addr_c 00000004 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c 00000060
jump_addr_a 00040028 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010 00000028
jump_addr_b 00040028 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010 00000028
jump_addr_c 00040028 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010 00000028
Outputs
firqp_a 0
firqp_b 0
firqp_c 0
finstruction_ 00000000 304bffff 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004
finstruction_ 00000000 304bffff 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004
finstruction_ 00000000 304bffff 344c0000 012a682a 0149702a 292f0004 29500004 8c110000 ac0b0004
fpc4_addr_a 00000000 00000040 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c
fpc4_addr_b 00000000 00000040 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c
fpc4_addr_c 00000000 00000040 00000044 00000048 0000004c 00000050 00000054 00000058 0000005c
fjump_addr_ 00000000 012ffffc 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010
fjump_addr_ 00000000 012ffffc 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010
fjump_addr_ 00000000 012ffffc 01300000 04a9a0a8 0525c0a8 04bc0010 05400010 00440000 002c0010

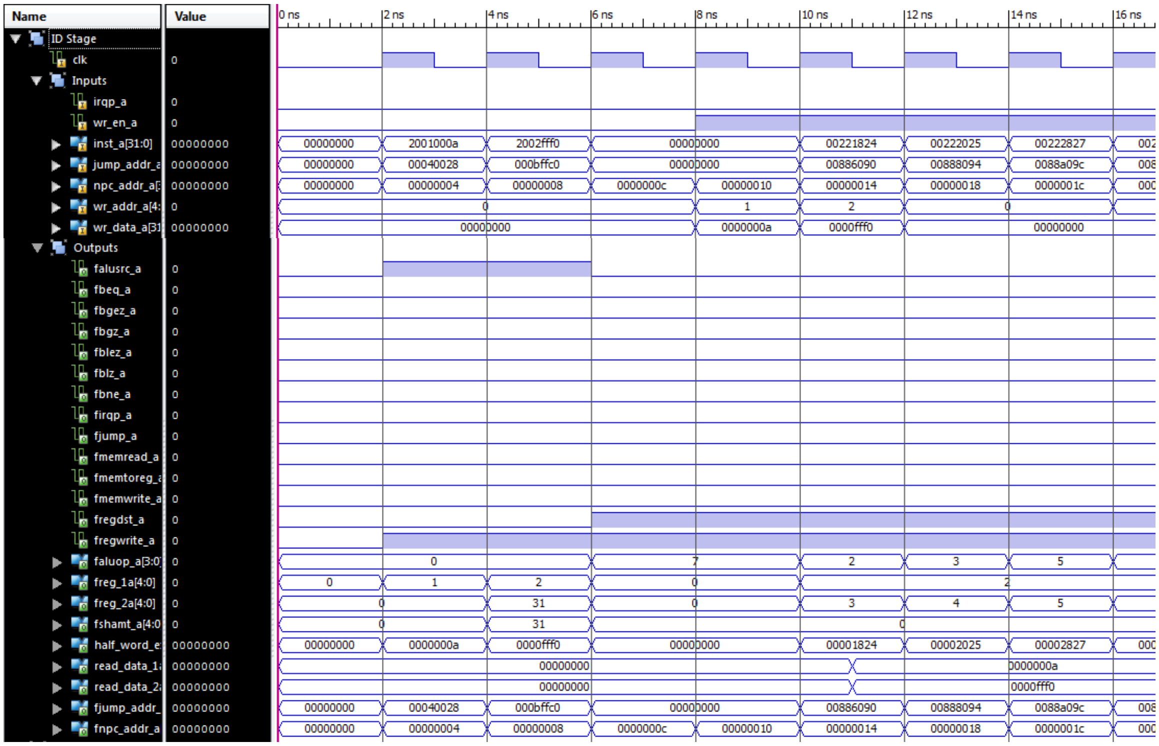

Name
Value
272 ns
274 ns
276 ns
278 ns
280 ns
282 ns
284 ns
286 ns
288 ns
IF/ID Register
clk 0
Inputs
irqp_a 0
irqp_b 0
irqp_c 0
instruction_ 2001000a
instruction_ 2001000a
instruction_ 2001000a
pc4_addr_a 00000004
pc4_addr_b 00000004
pc4_addr_c 00000004
jump_addr_ 00040028
jump_addr_ 00040028
jump_addr_ 00040028
Outputs
firqp_a 0
firqp_b 0
firqp_c 0
finstruction_ 00000000
finstruction_ 00000000
finstruction_ 00000000
fpc4_addr_a 00000000
fpc4_addr_b 00000000
fpc4_addr_c 00000000
fjump_addr_ 00000000
fjump_addr_ 00000000
fjump_addr_ 00000000

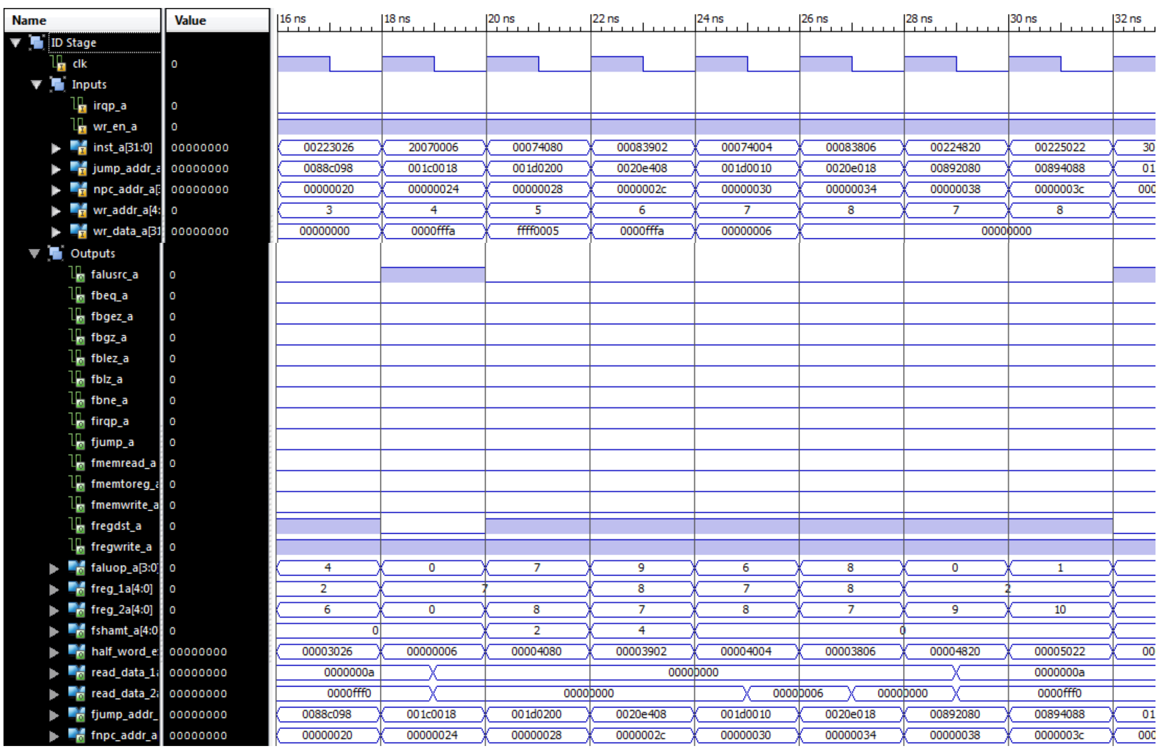

Name
Value
288 ns
290 ns
292 ns
294 ns
296 ns
298 ns
300 ns
302 ns
304 ns
IF/ID Register
clk 0
Inputs
irqp_a 0
irqp_b 0
irqp_c 0
instruction_ 2001000a
instruction_ 2001000a
instruction_ 2001000a
pc4_addr_a 00000004
pc4_addr_b 00000004
pc4_addr_c 00000004
jump_addr_ 00040028
jump_addr_ 00040028
jump_addr_ 00040028
Outputs
firqp_a 0
firqp_b 0
firqp_c 0
finstruction_ 00000000
finstruction_ 00000000
finstruction_ 00000000
fpc4_addr_a 00000000
fpc4_addr_b 00000000
fpc4_addr_c 00000000
fjump_addr_ 00000000
fjump_addr_ 00000000
fjump_addr_ 00000000

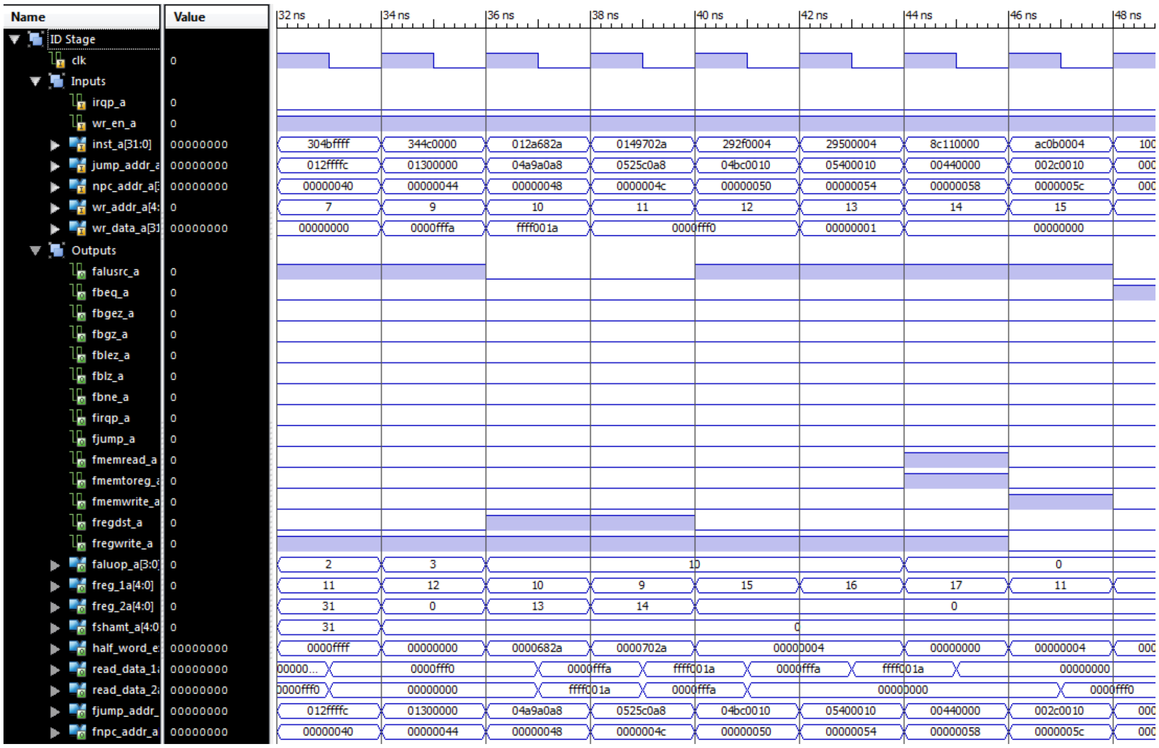


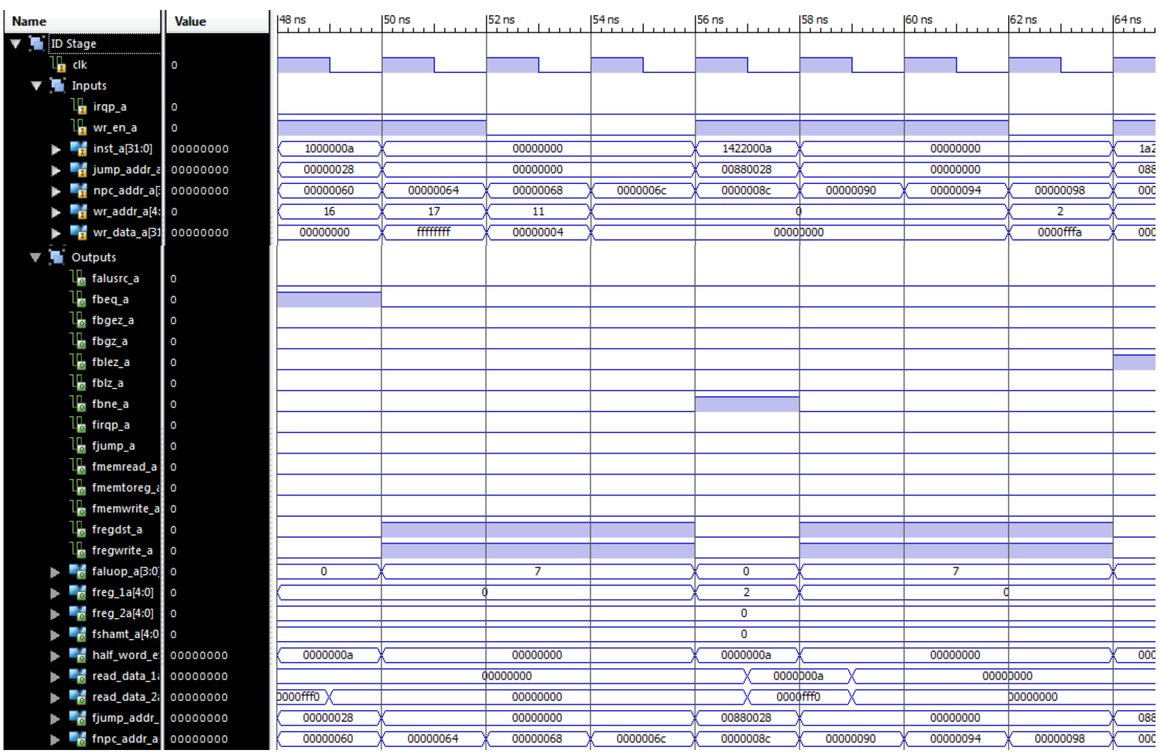


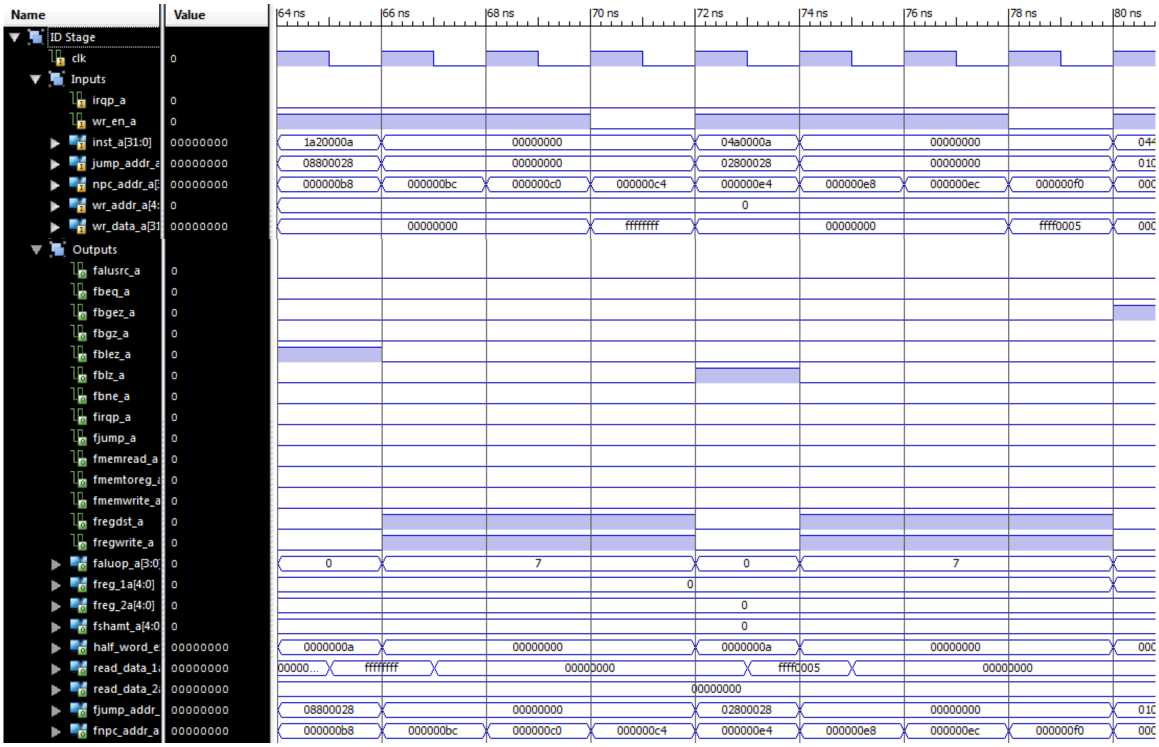
F. ID STAGE

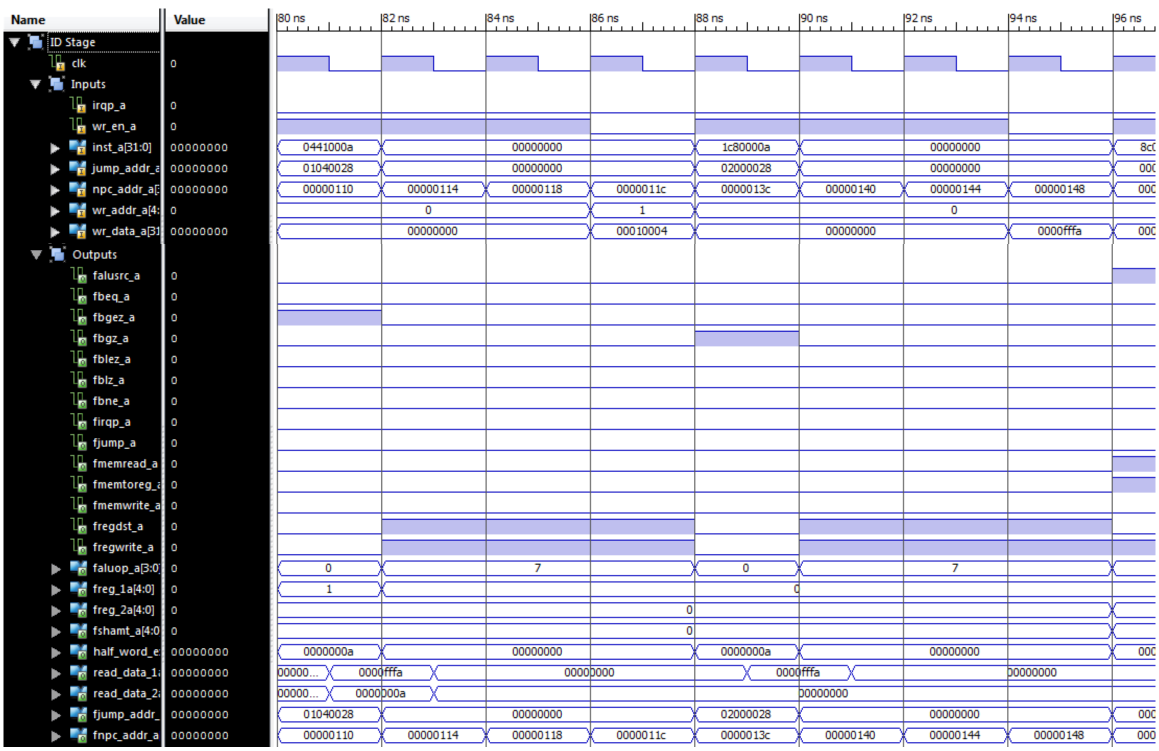


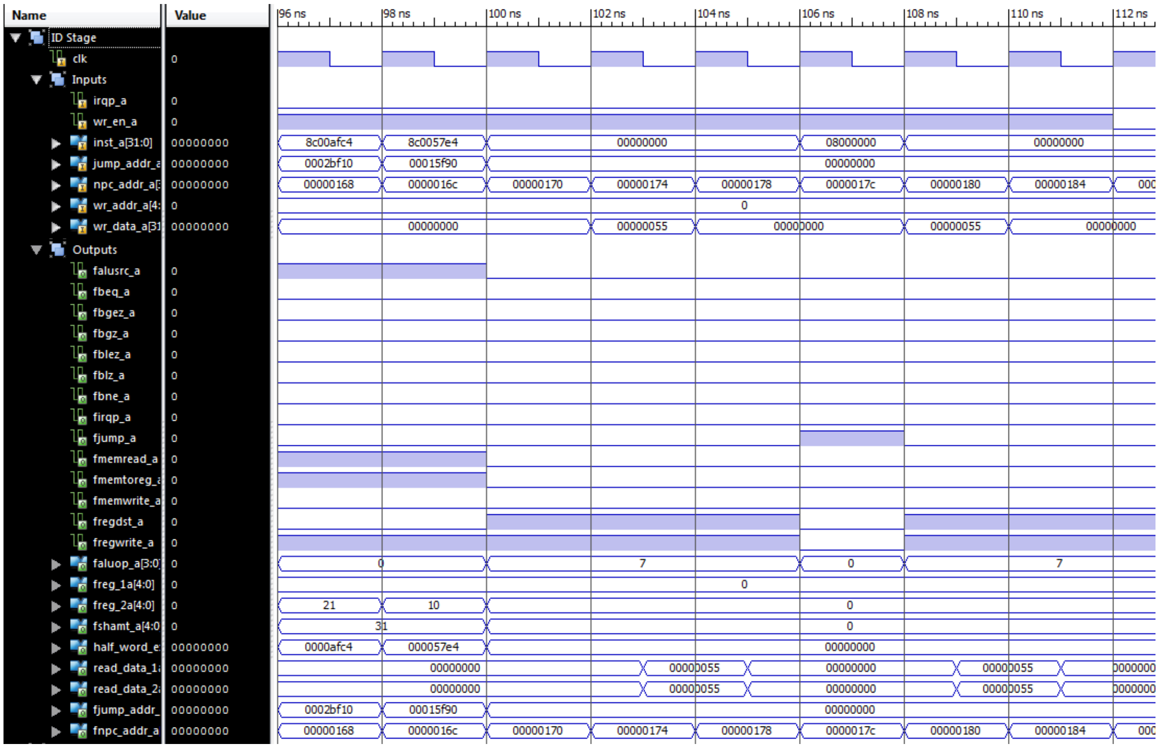


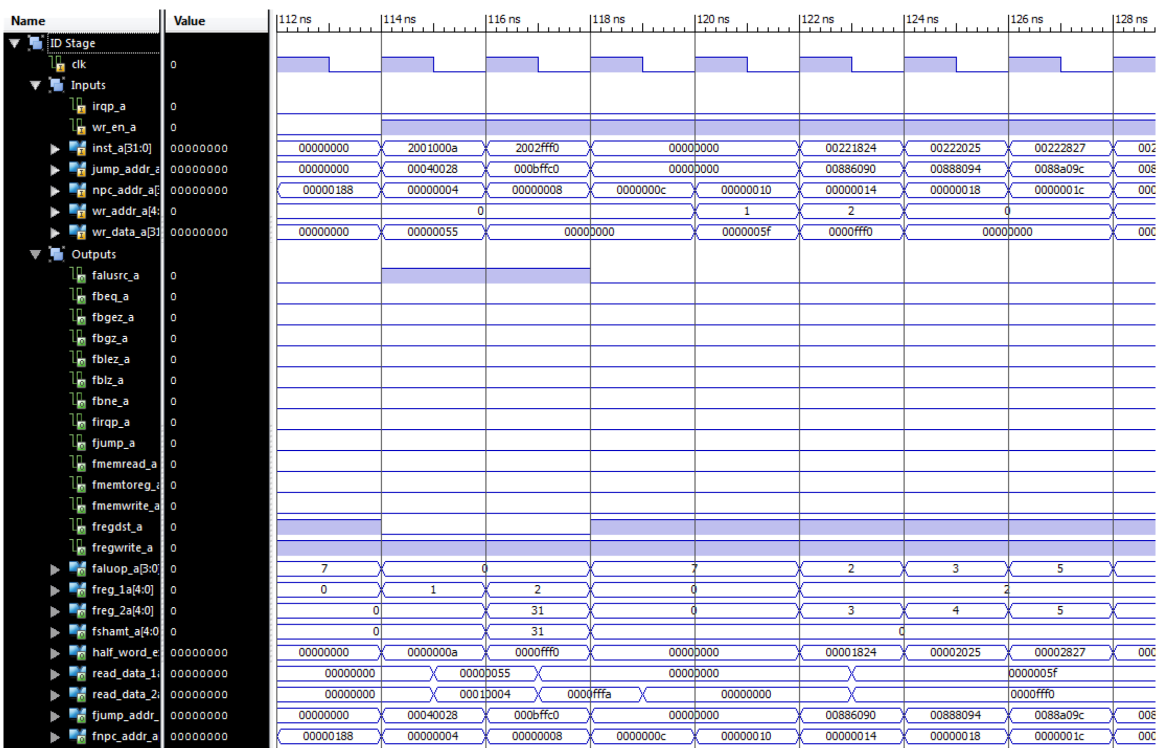


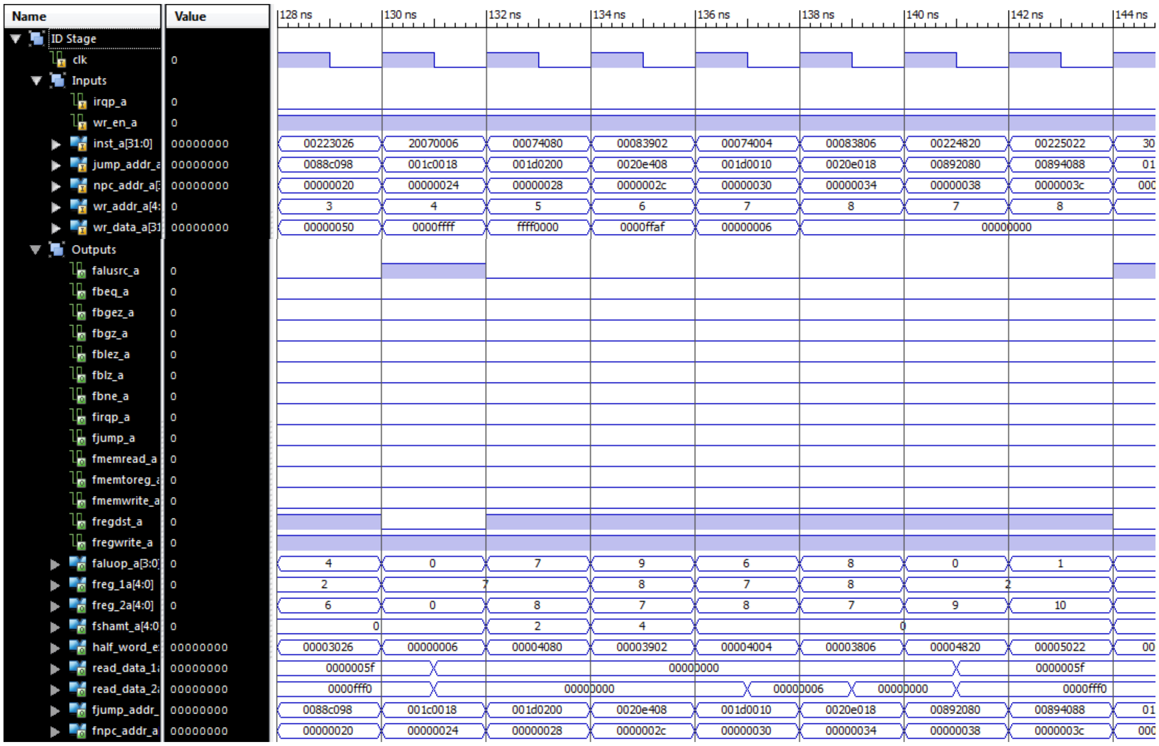


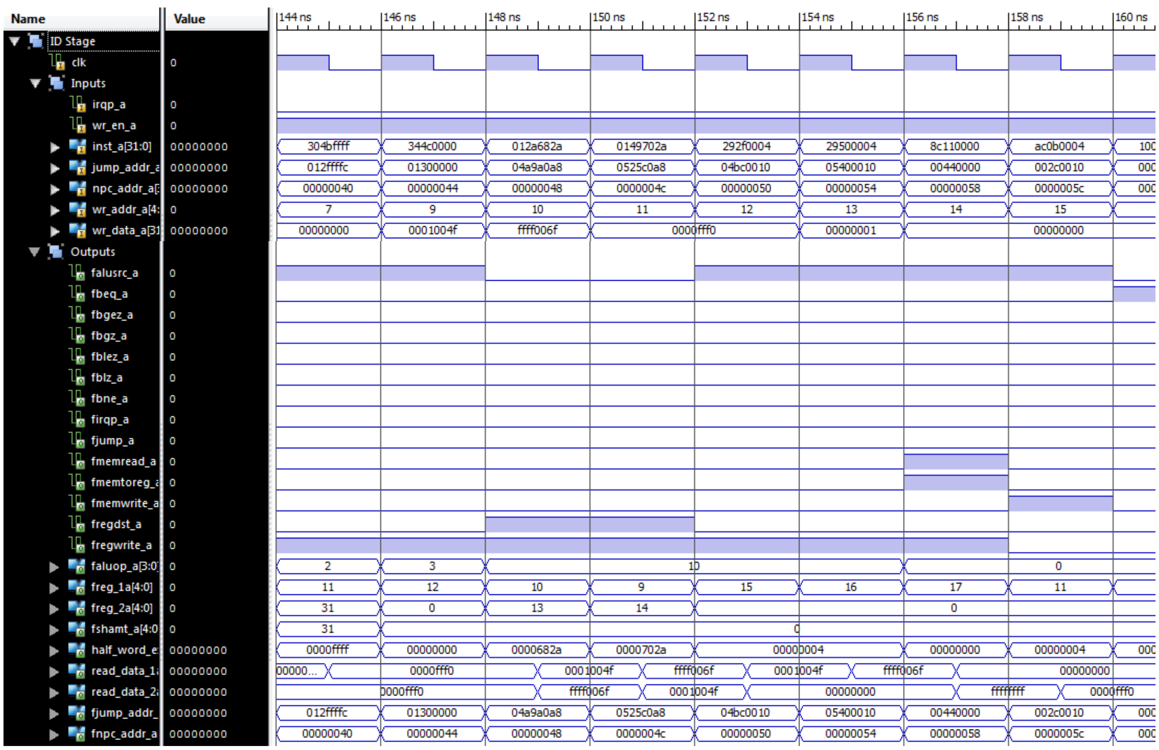


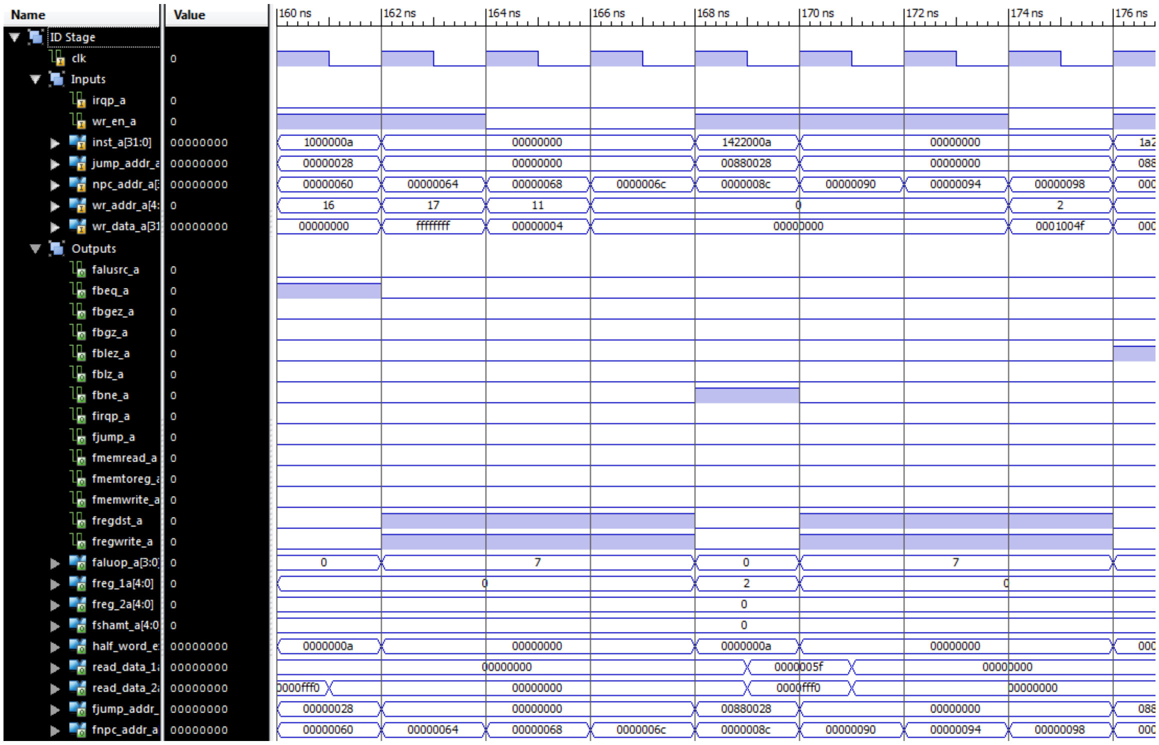


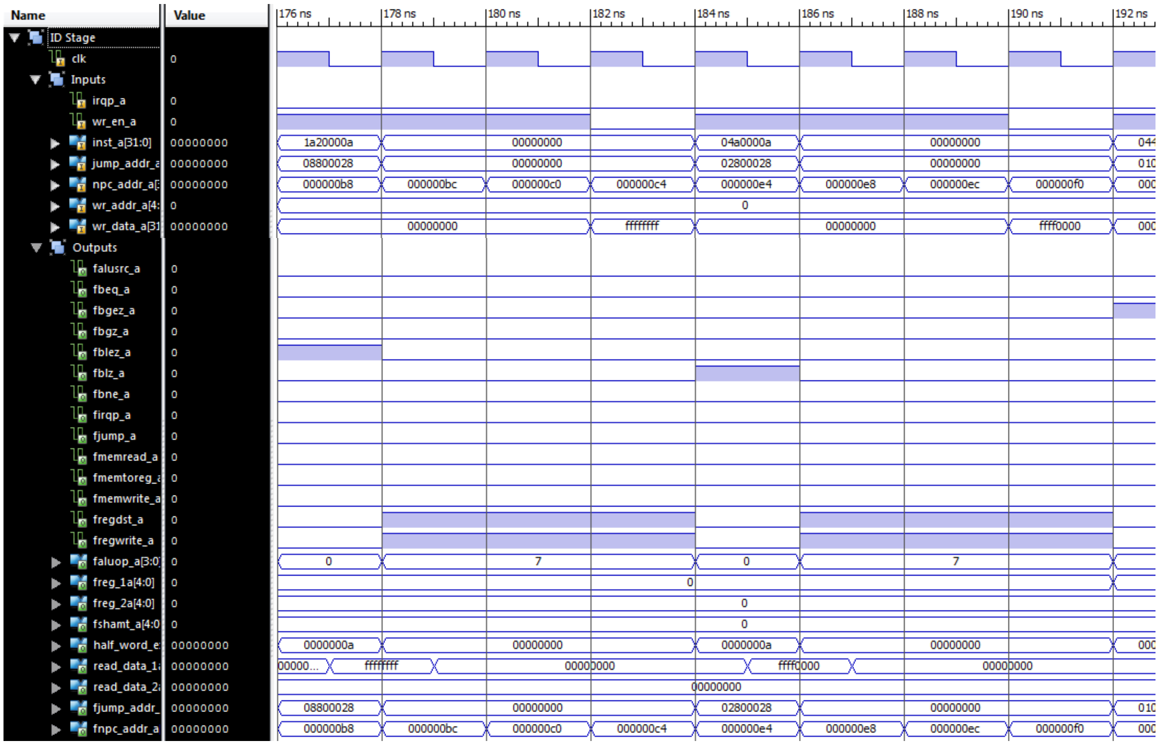


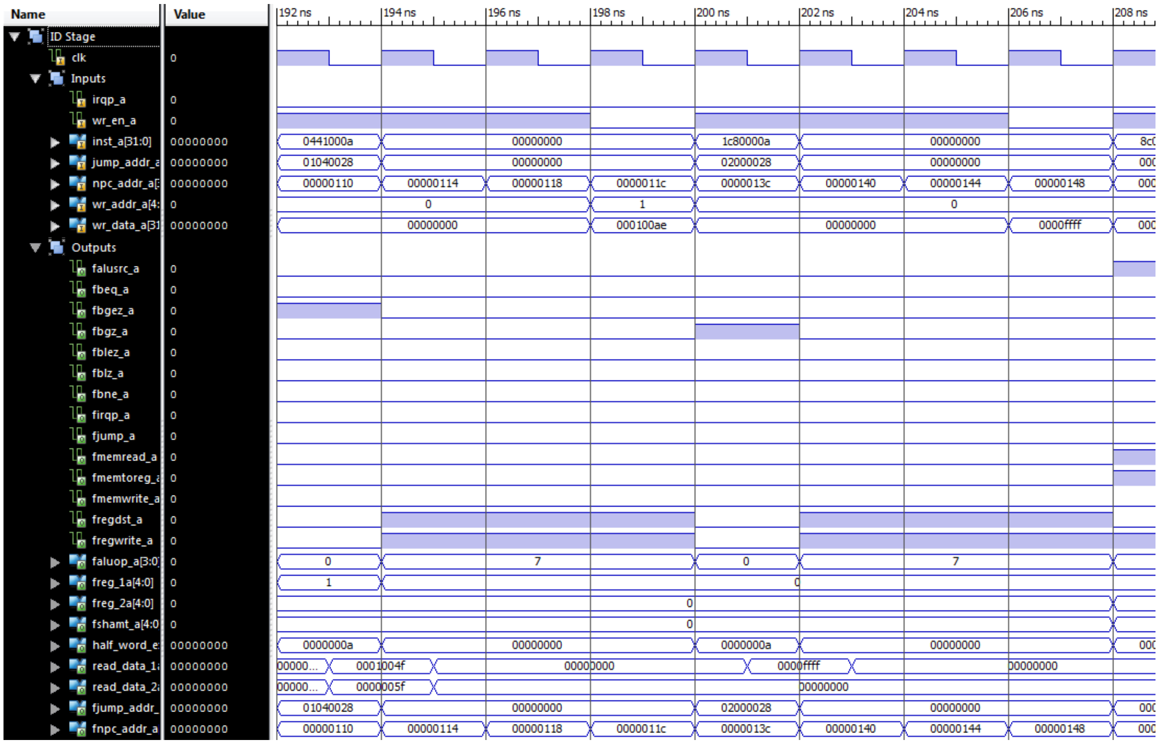


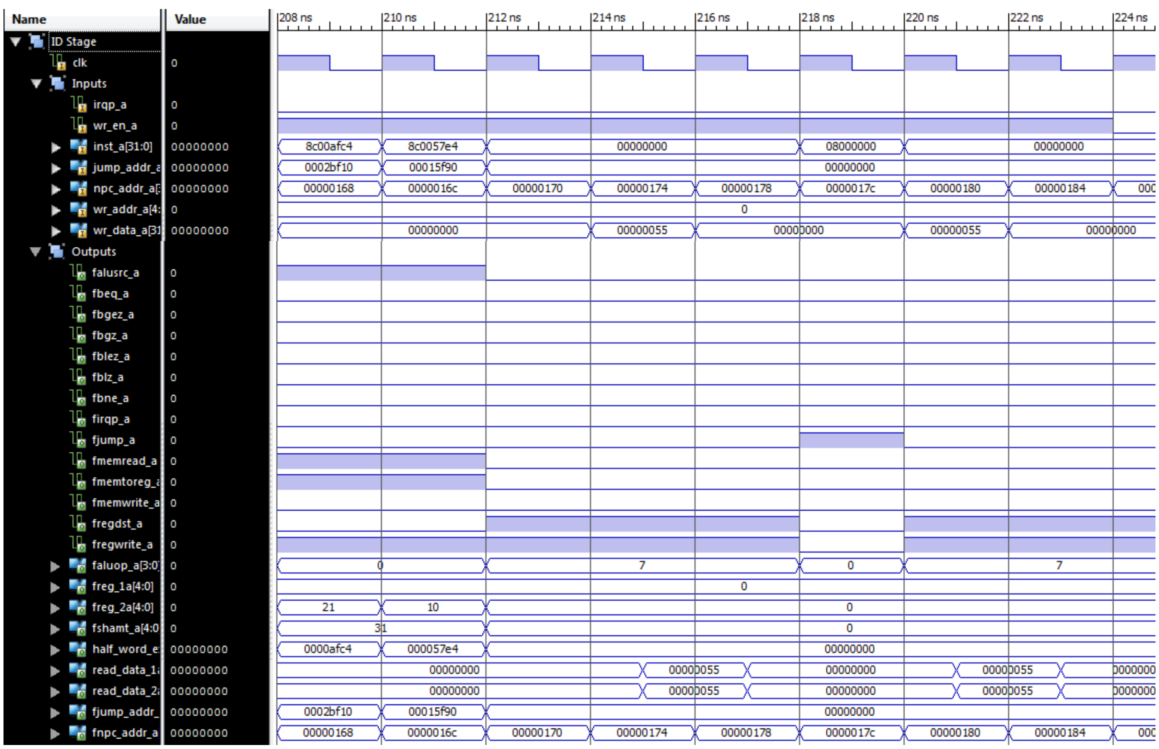


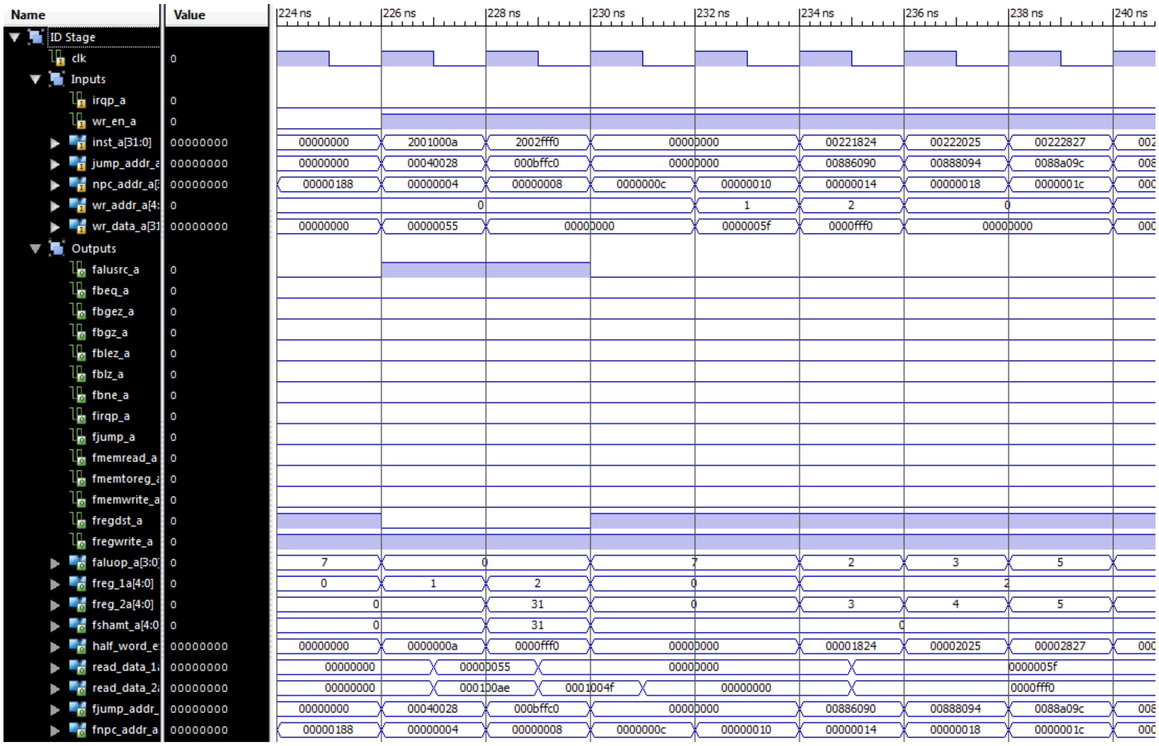


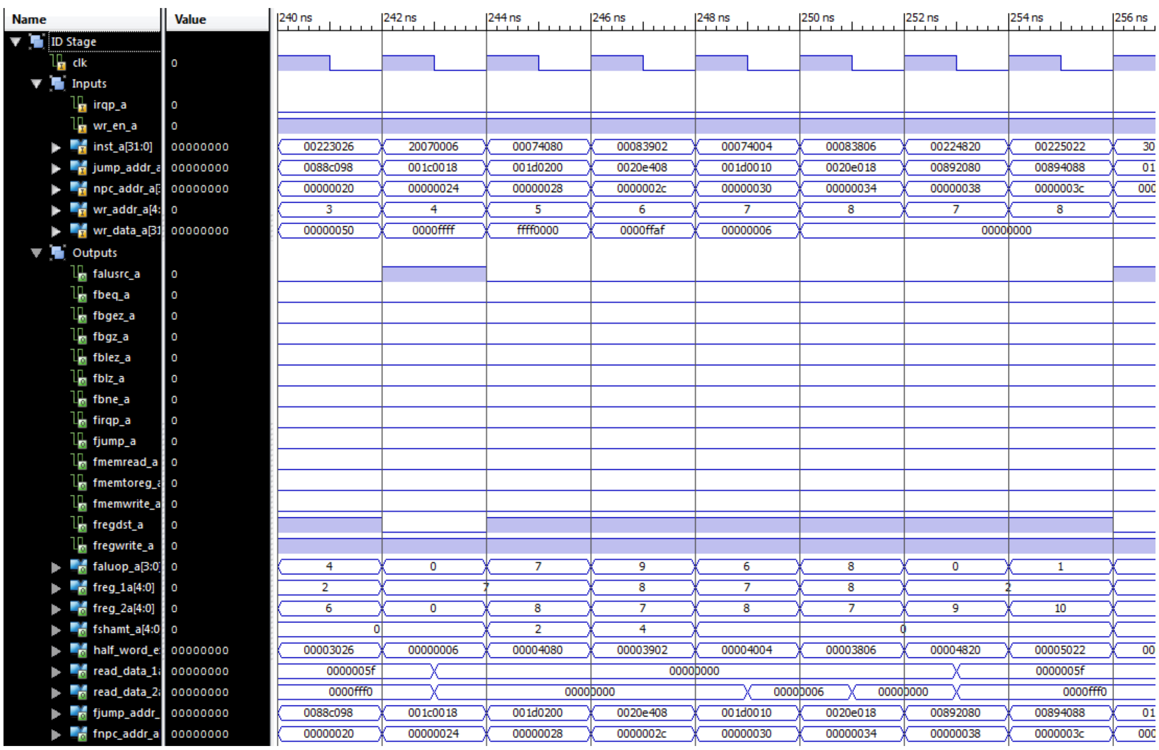


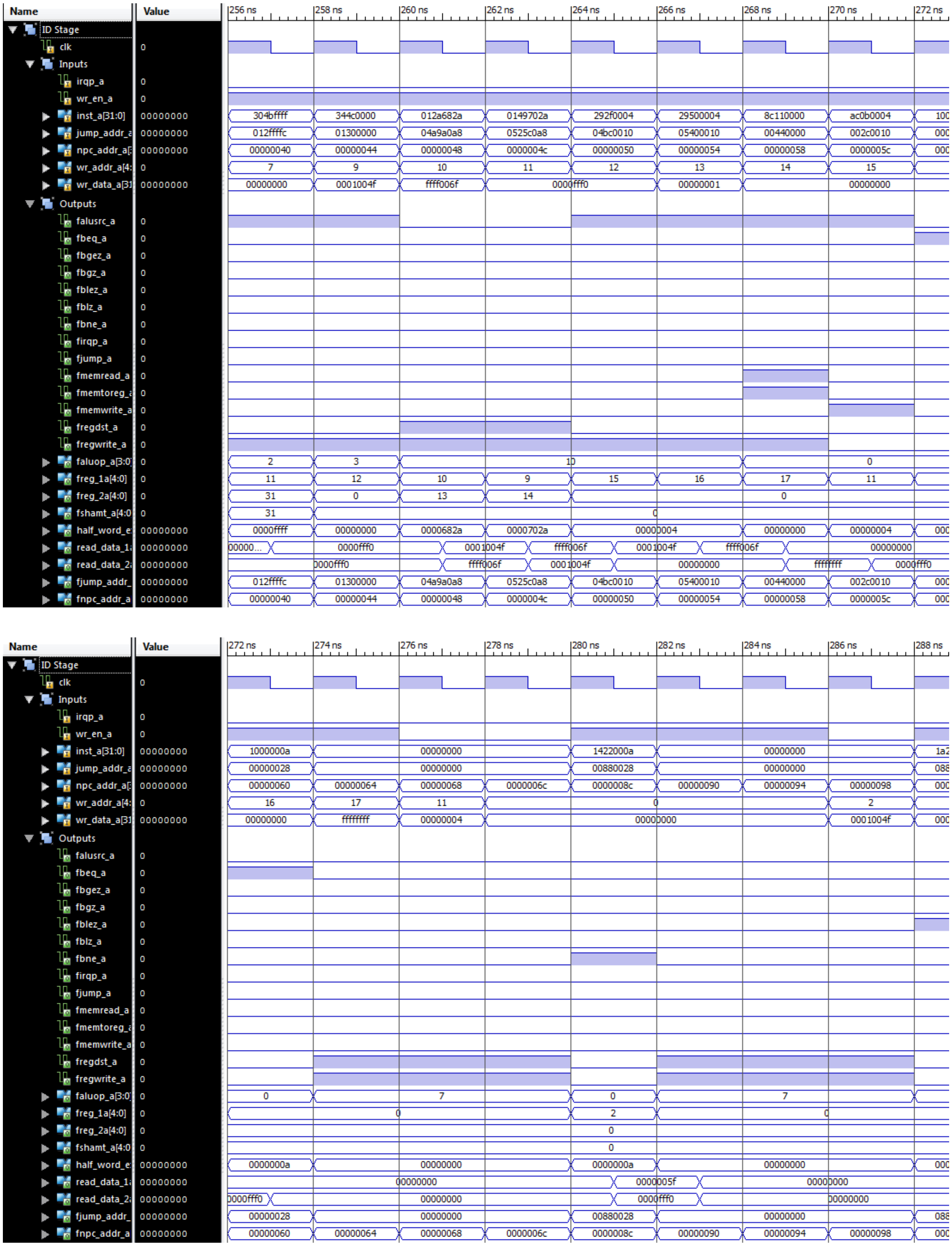


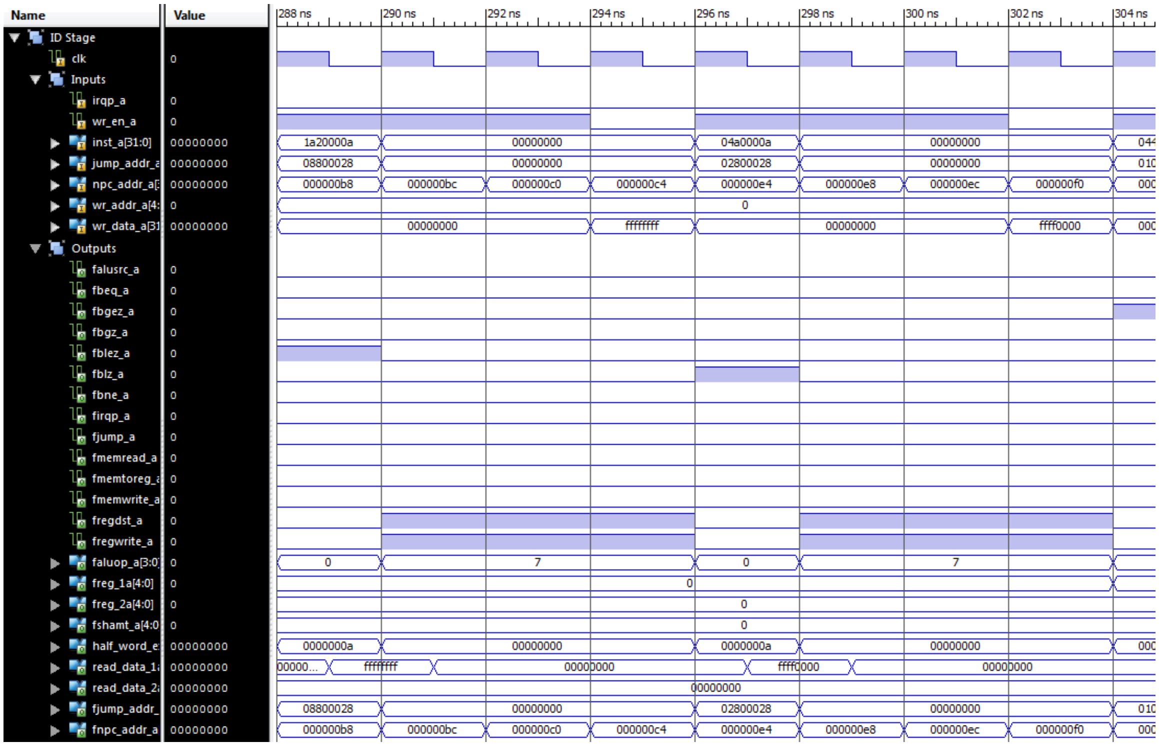

Name
Value
ID Stage
clk
Inputs
irqp_a
wr_en_a
inst_a[31:0]
jump_addr_a
npc_addr_a
wr_addr_a[4:
wr_data_a[31
Outputs
falusrc_a
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregdst_a
fregwrite_a
faluop_a[3:0]
freg_1a[4:0]
freg_2a[4:0]
fshamt_a[4:0]
half_word_e
read_data_1
read_data_2
fjump_addr_
fnpc_addr_a

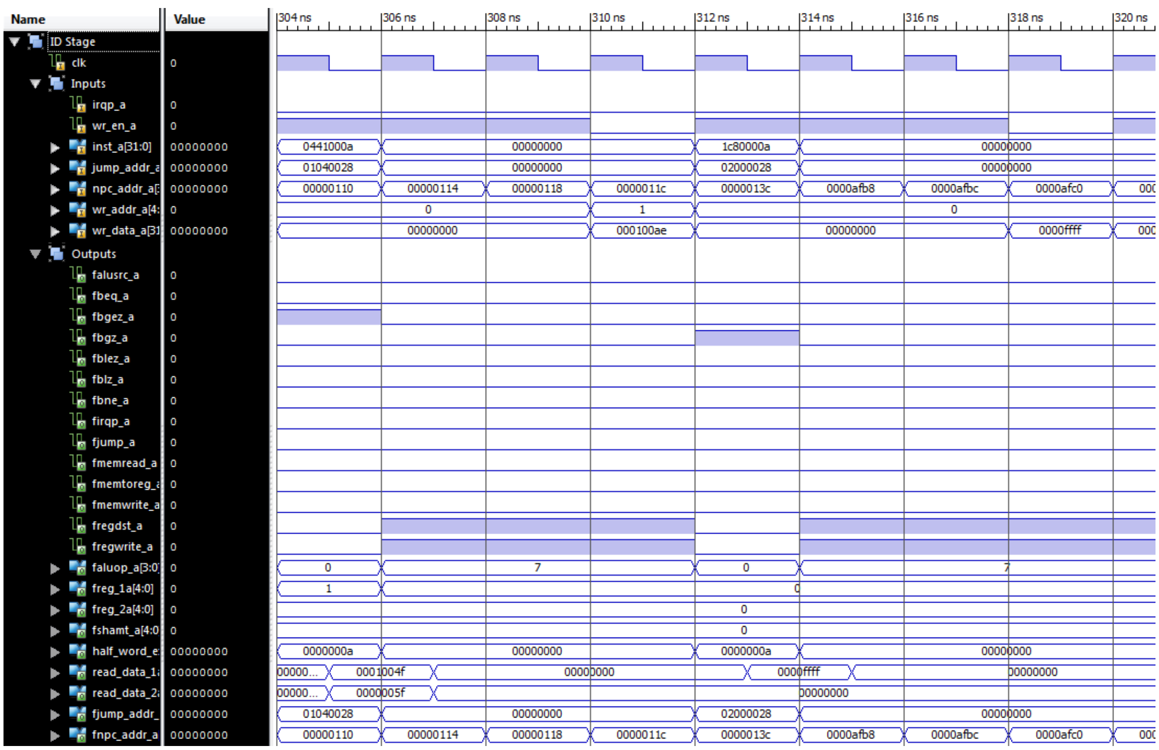

Name
Value
ID Stage
clk
Inputs
irqp_a
wr_en_a
inst_a[31:0]
jump_addr_a
npc_addr_a
wr_addr_a[4:
wr_data_a[31
Outputs
falusrc_a
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregdst_a
fregwrite_a
faluop_a[3:0]
freg_1a[4:0]
freg_2a[4:0]
fshamt_a[4:0]
half_word_e
read_data_1
read_data_2
fjump_addr_
fnpc_addr_a

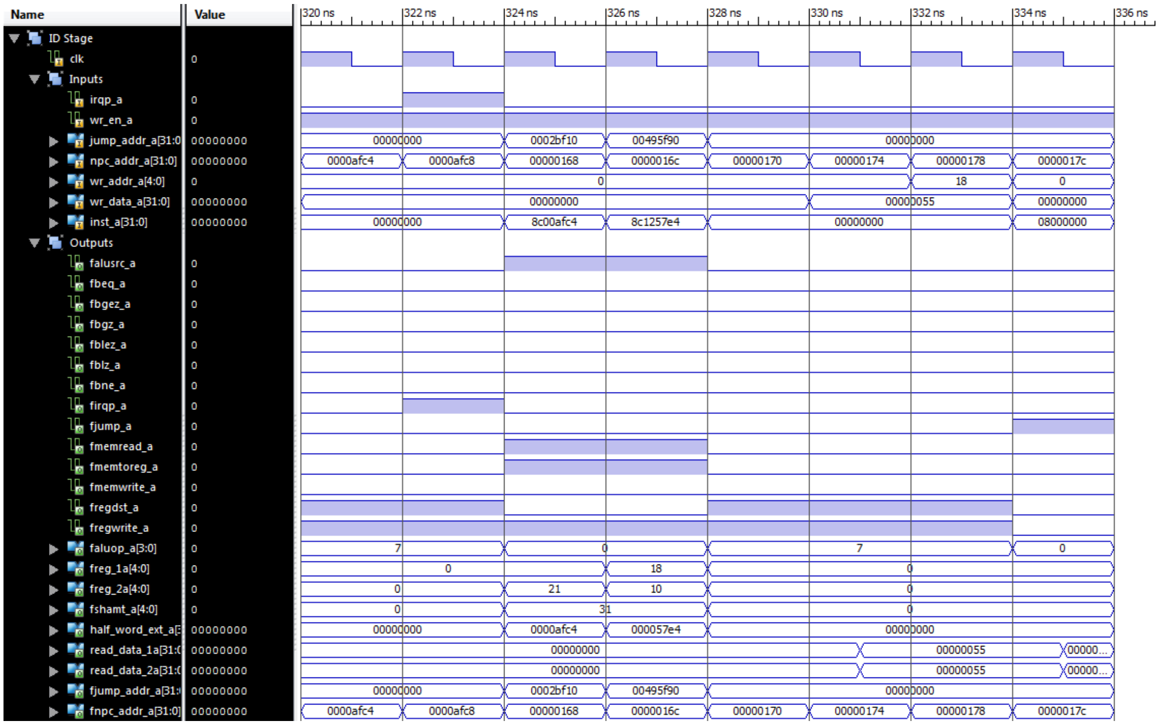




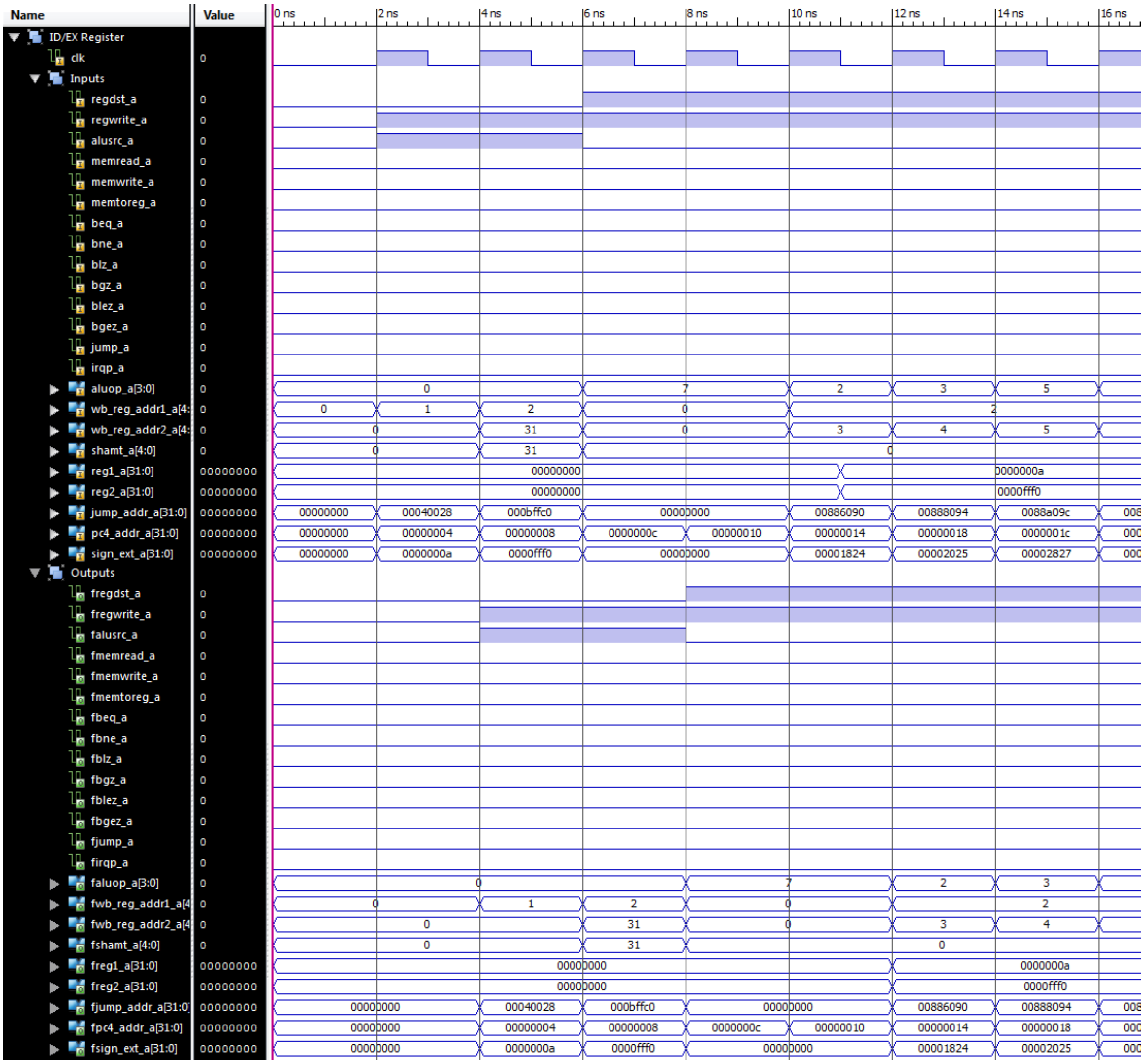


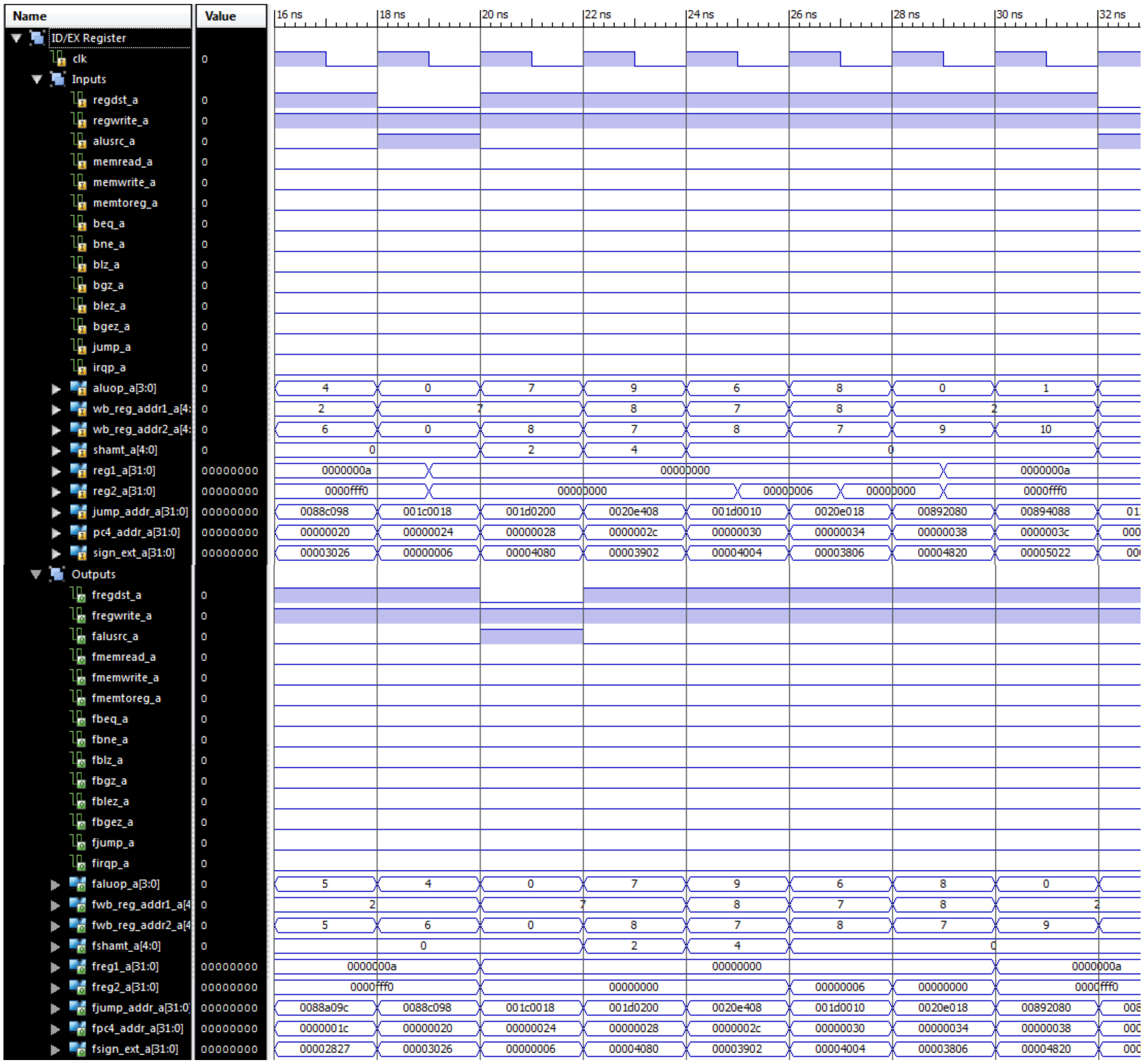


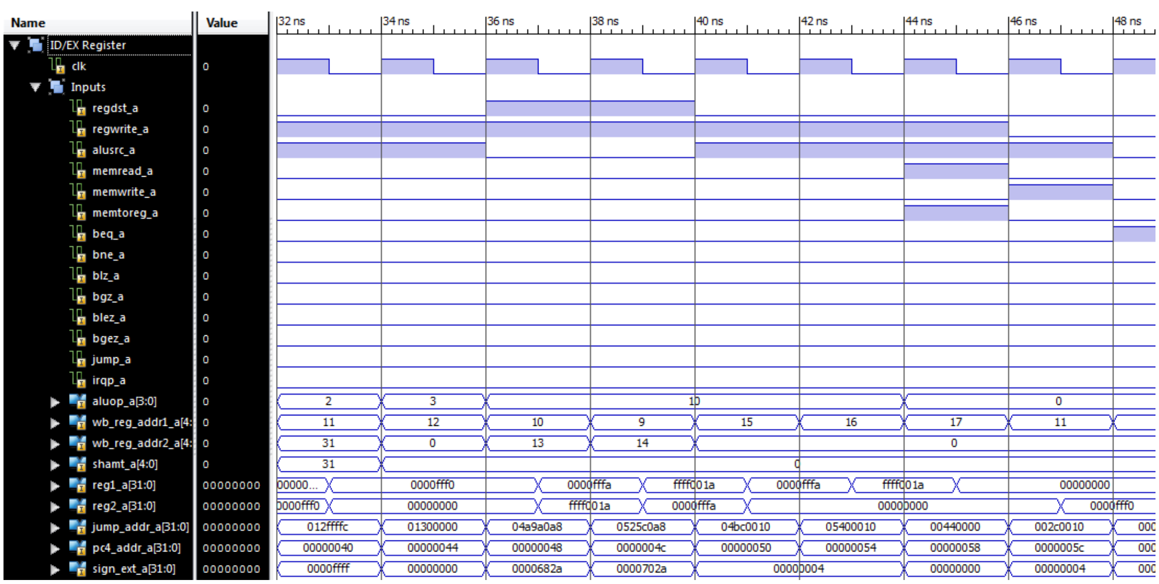


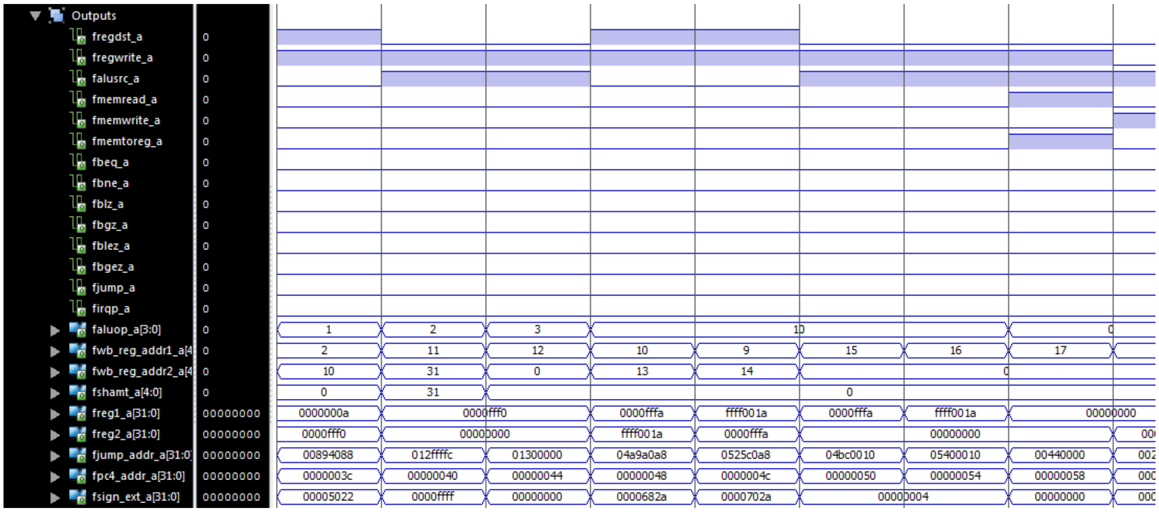

Name
Value
ID Stage
clk
Inputs
irqp_a
wr_en_a
jump_addr_a[31:0]
npc_addr_a[31:0]
wr_addr_a[4:0]
wr_data_a[31:0]
inst_a[31:0]
Outputs
falusrc_a
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregdst_a
fregwrite_a
faluop_a[3:0]
freg_1a[4:0]
freg_2a[4:0]
fshamt_a[4:0]
fjump_addr_a[31:0]
fnpc_addr_a[31:0]
320 ns
322 ns
324 ns
326 ns
328 ns
330 ns
332 ns
334 ns
336 ns
0002bf10
00495f90
0000afc4
0000afc8
00000168
0000016c
00000170
00000174
00000178
0000017c
00000055
8c00afc4
8c1257e4
08000000
000057e4

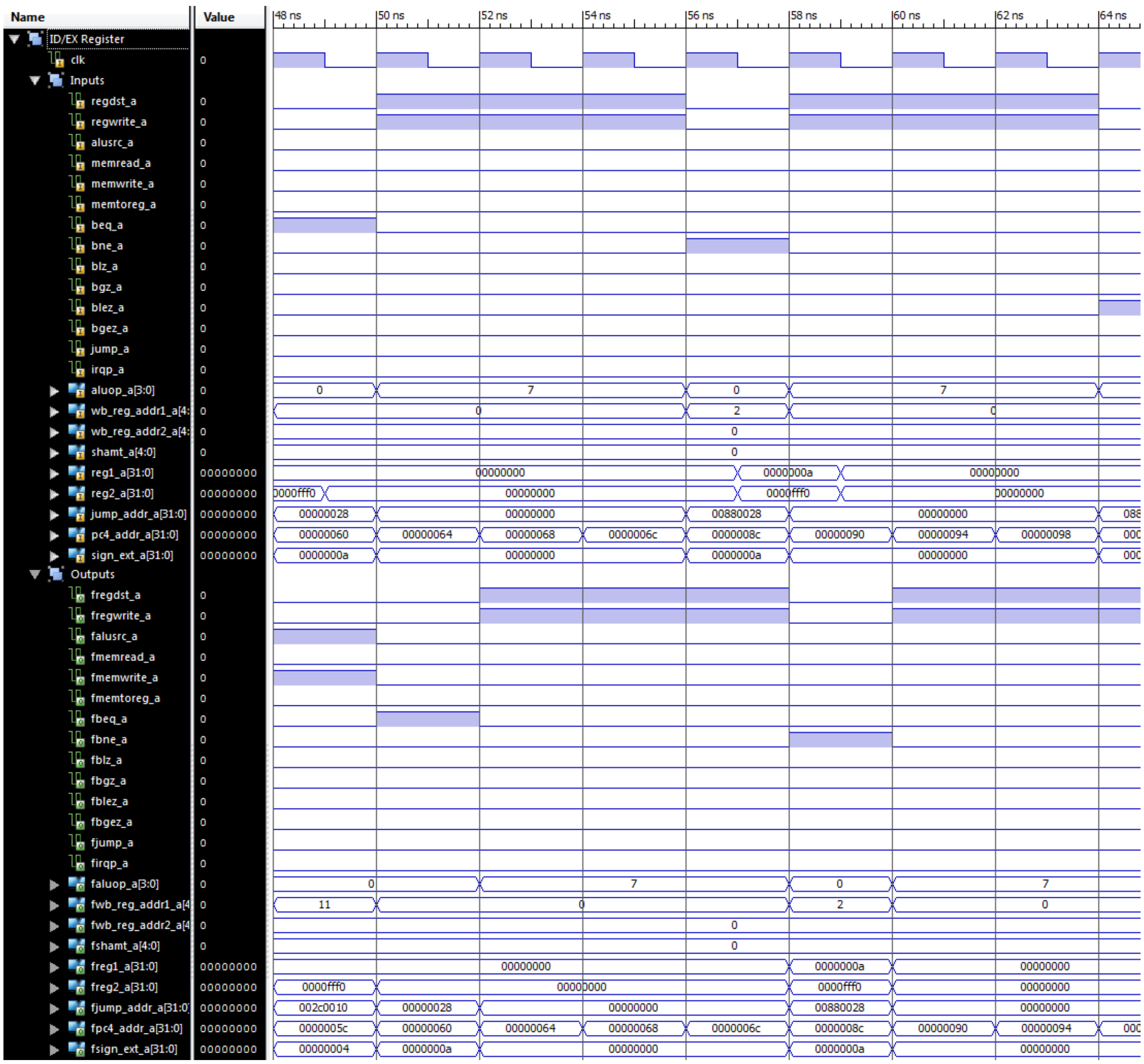
G. ID/EX REGISTER

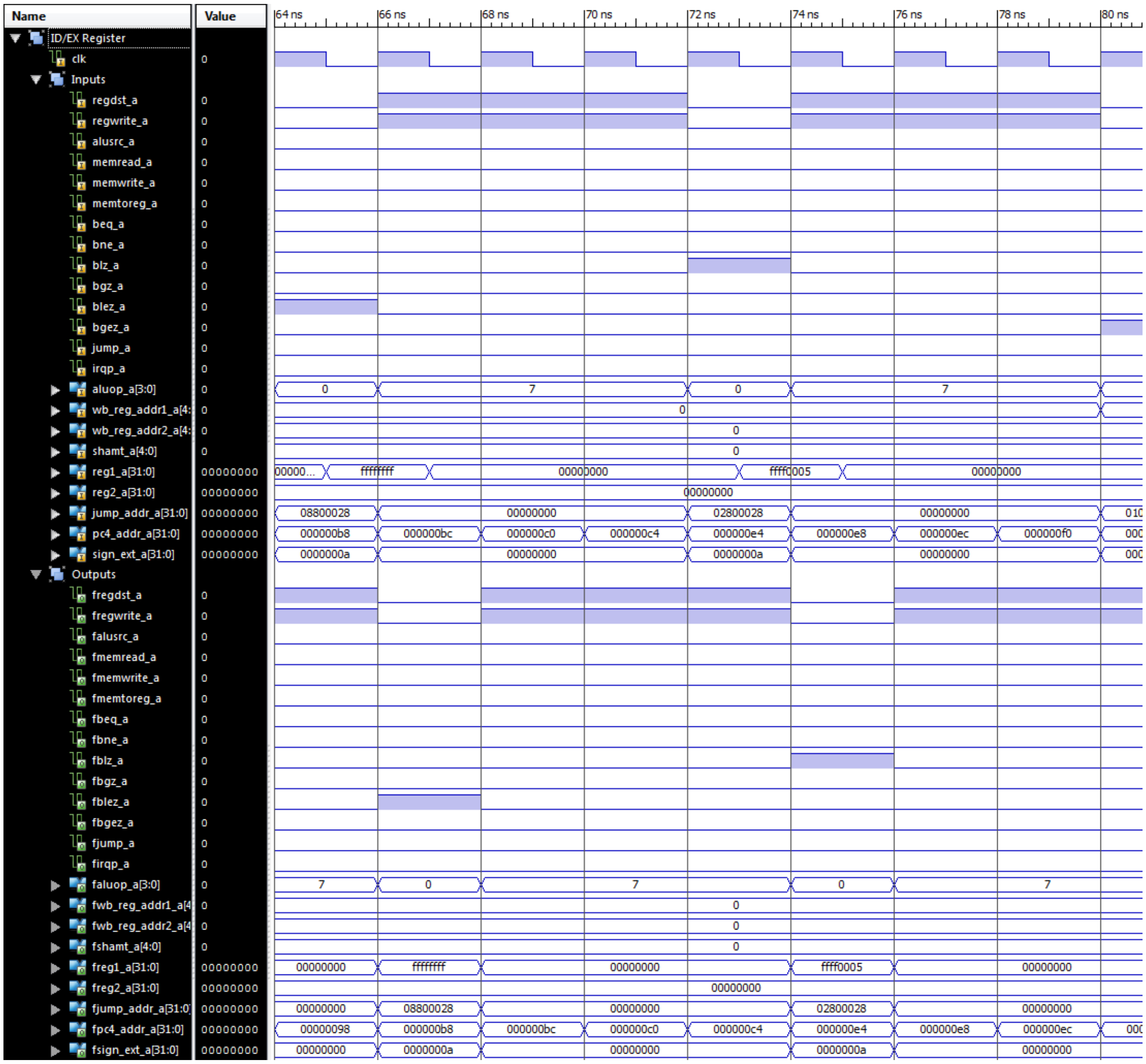


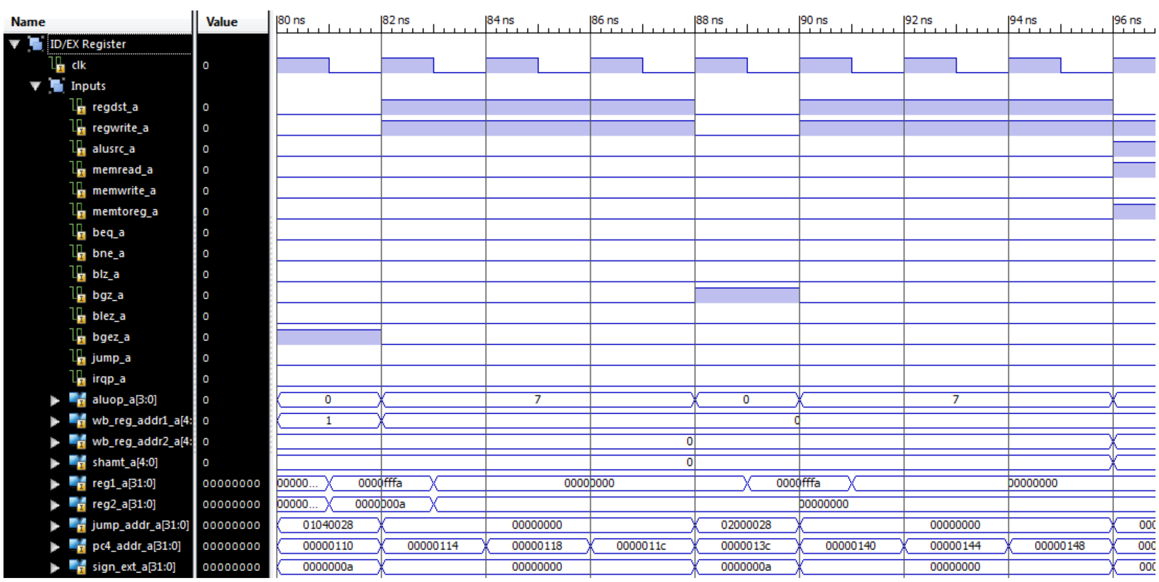


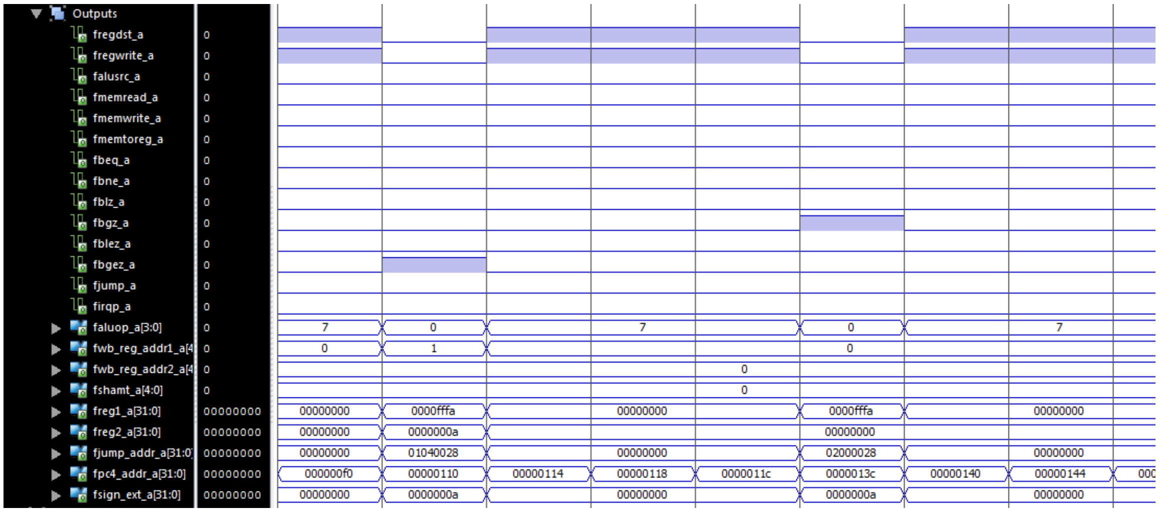


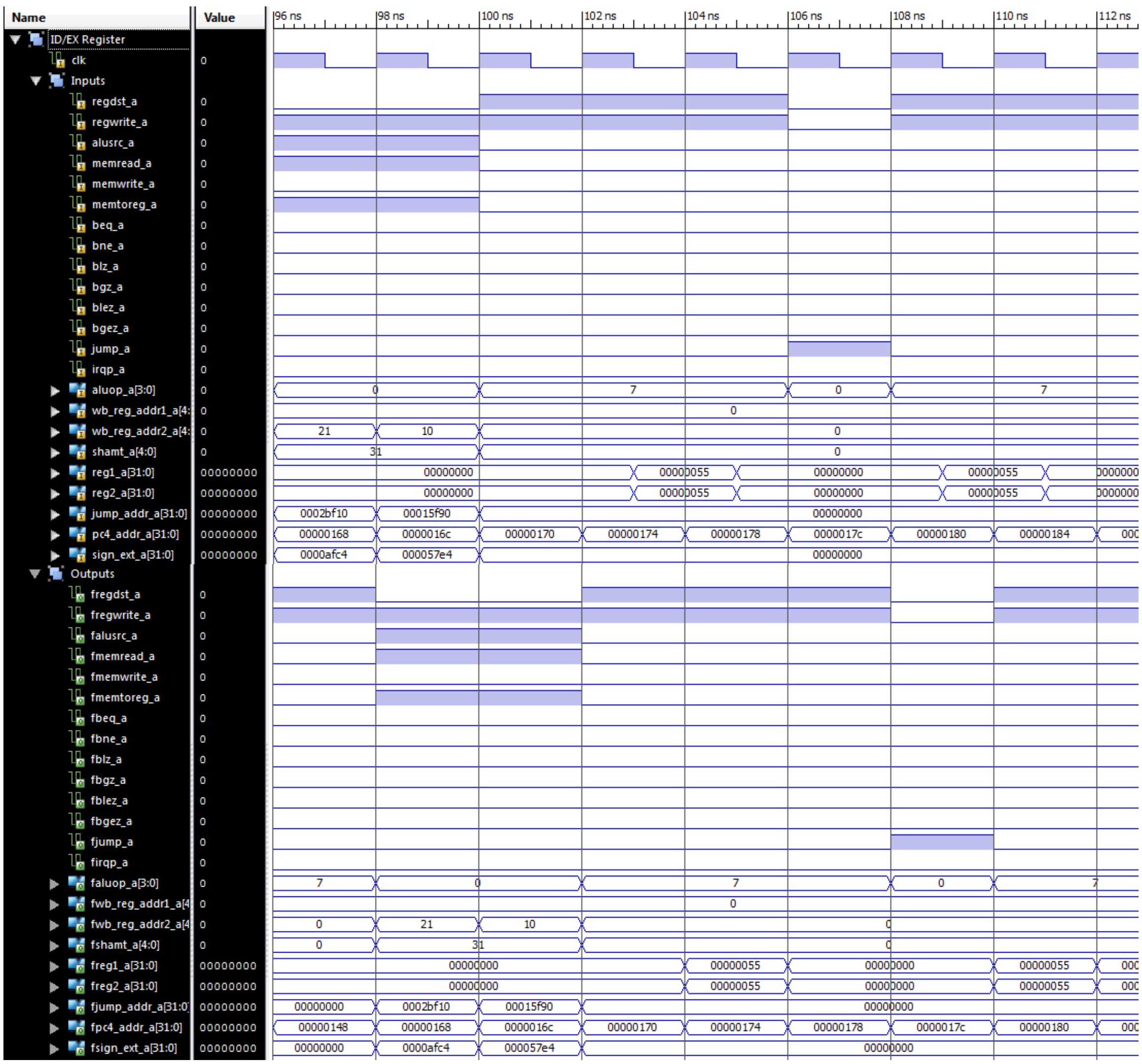


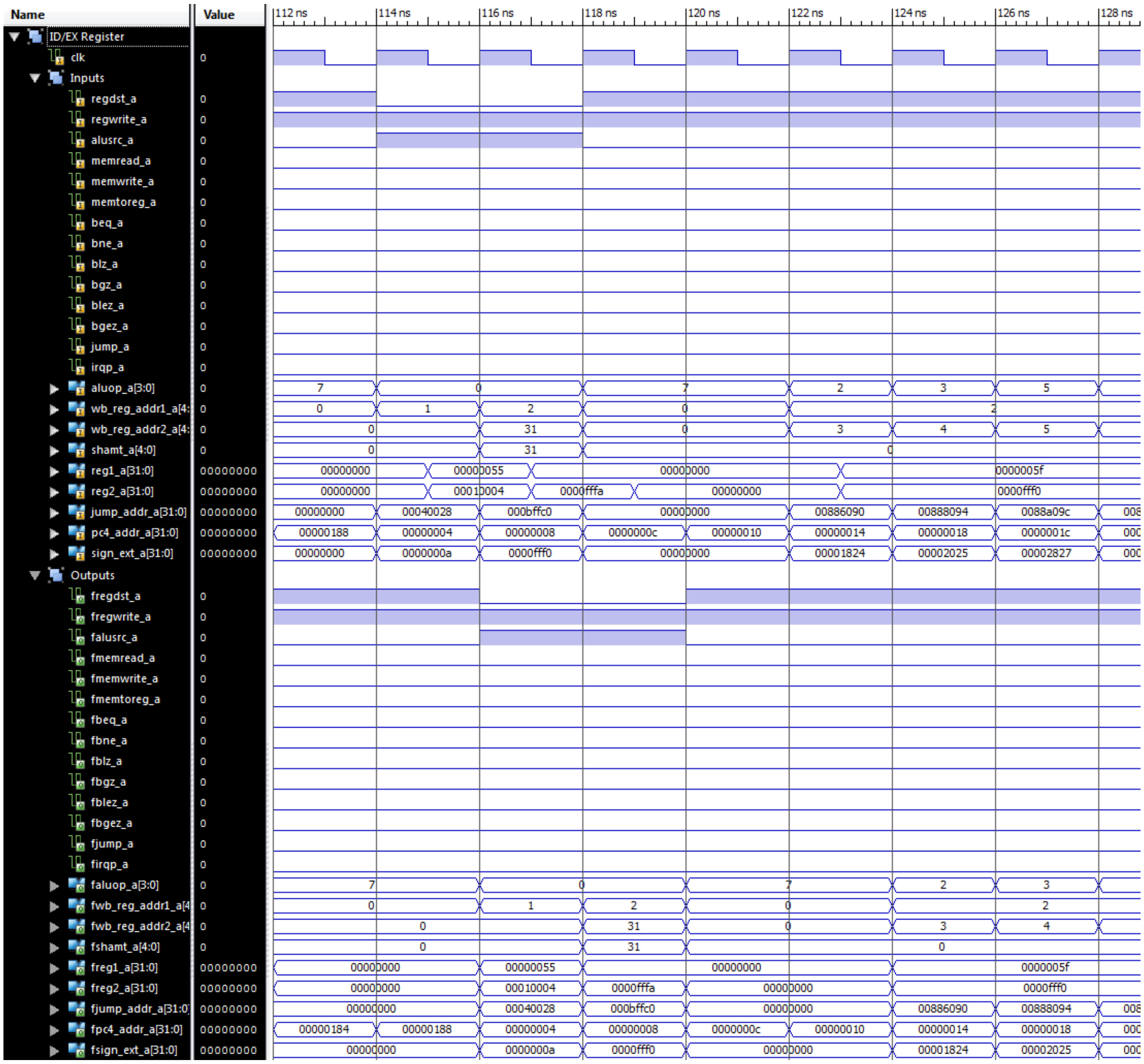


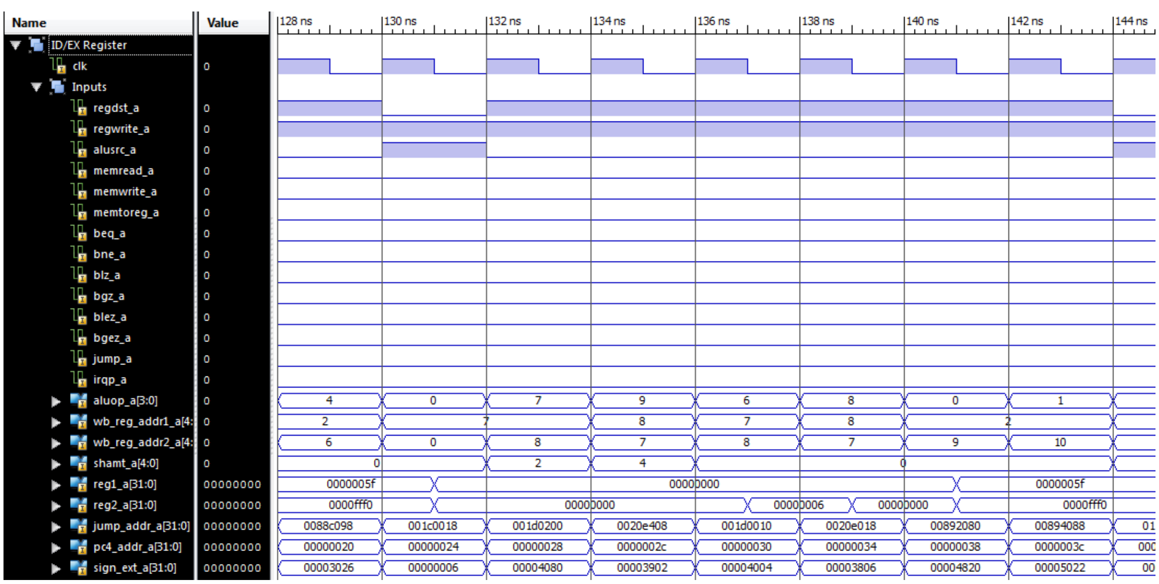


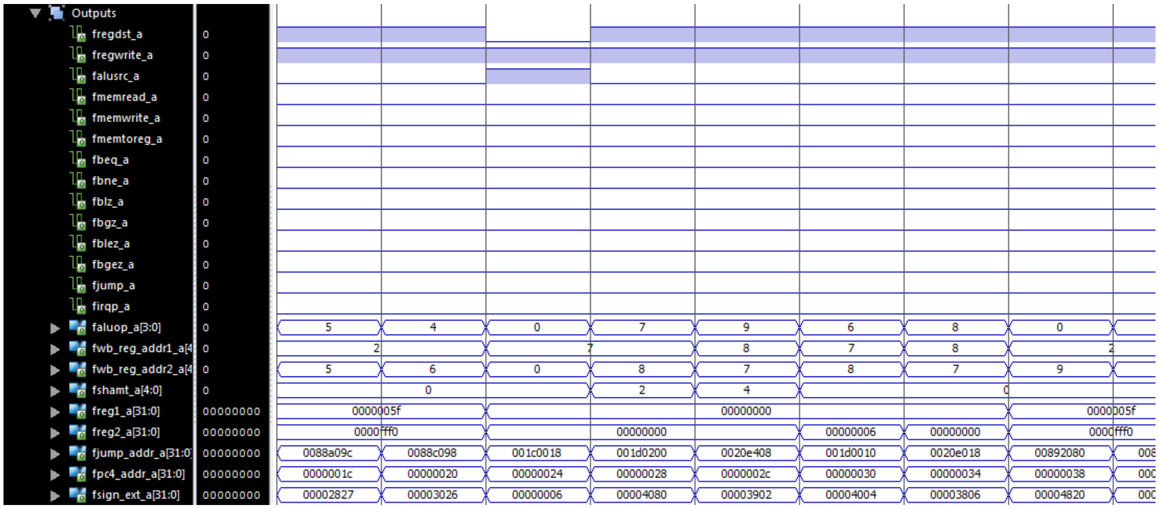


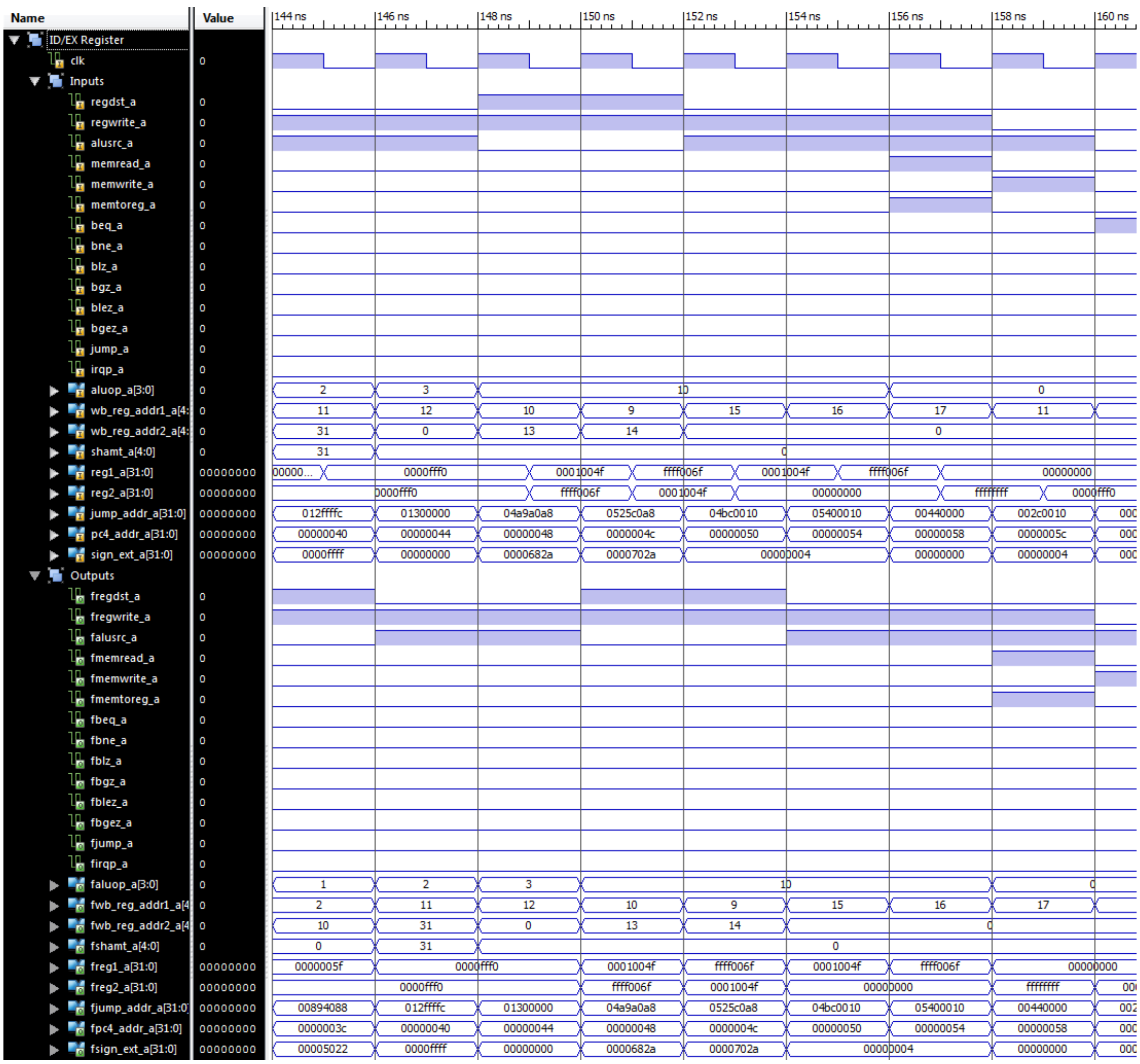

Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

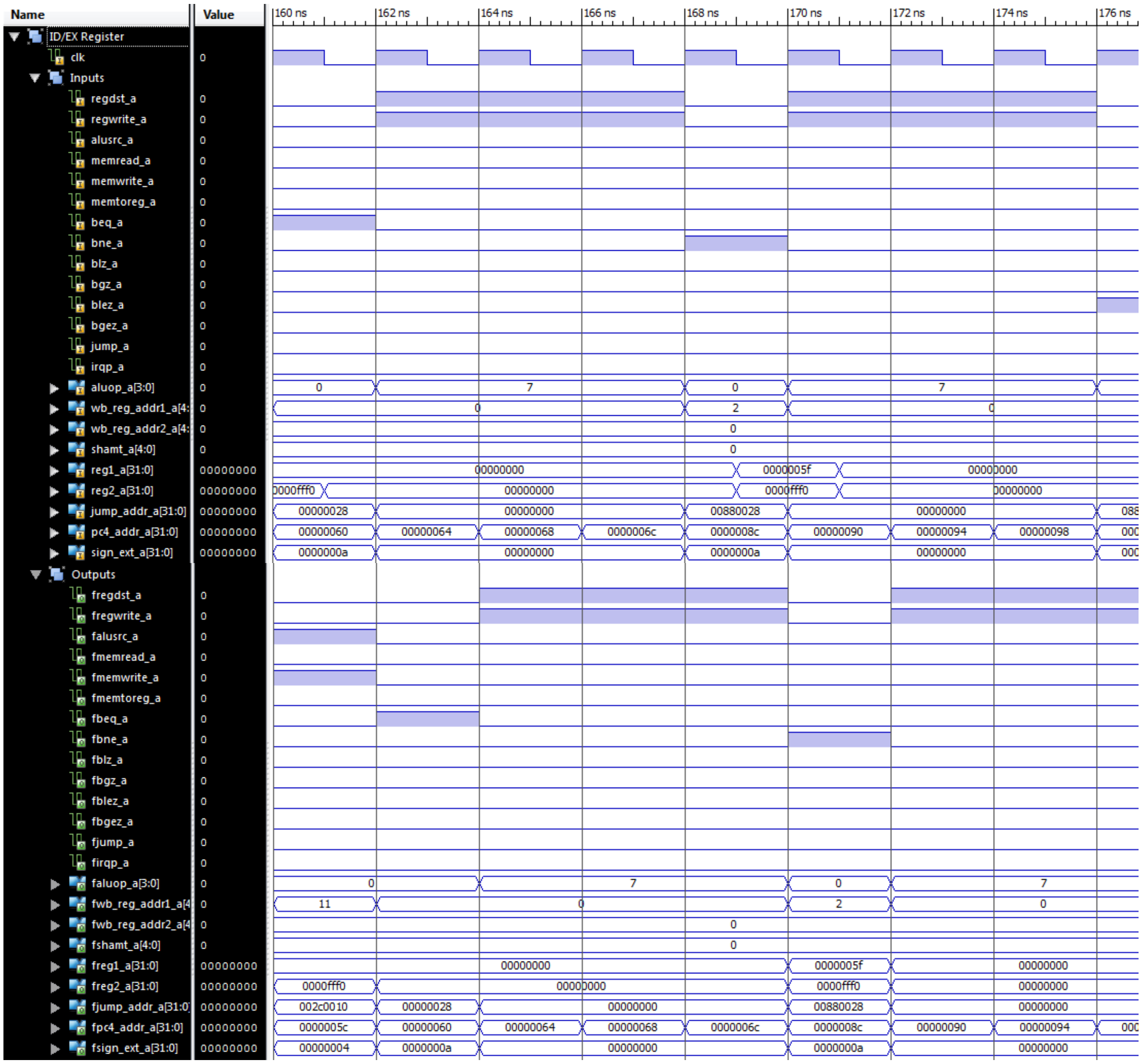

Name
Value
96 ns
98 ns
100 ns
102 ns
104 ns
106 ns
108 ns
110 ns
112 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]
Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

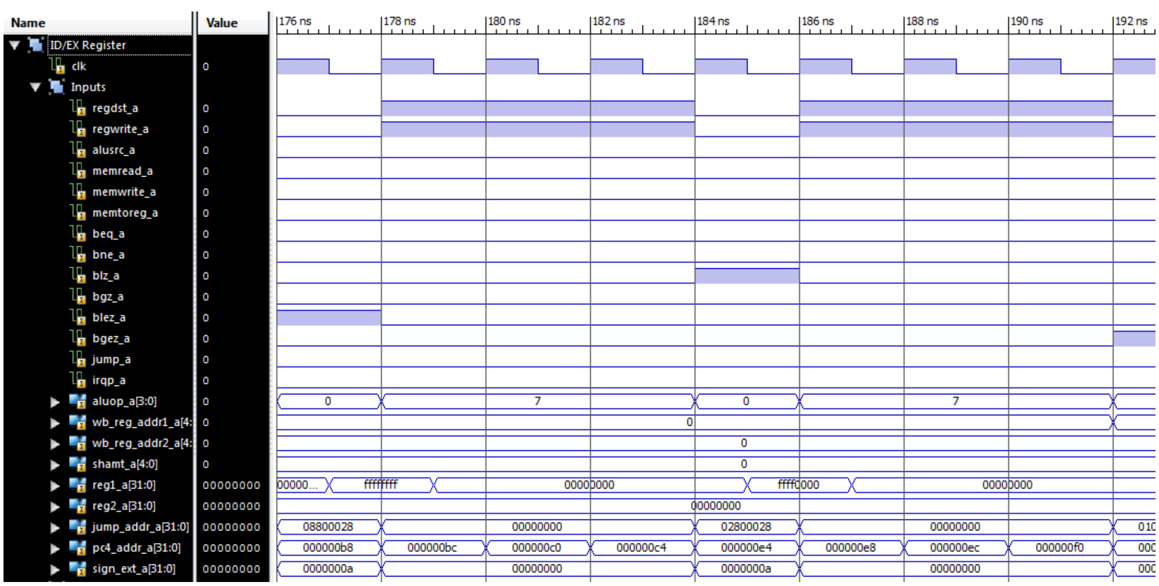

Name
Value
112 ns
114 ns
116 ns
118 ns
120 ns
122 ns
124 ns
126 ns
128 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]
Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fwb_reg_addr1_a[4
fwb_reg_addr2_a[4
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

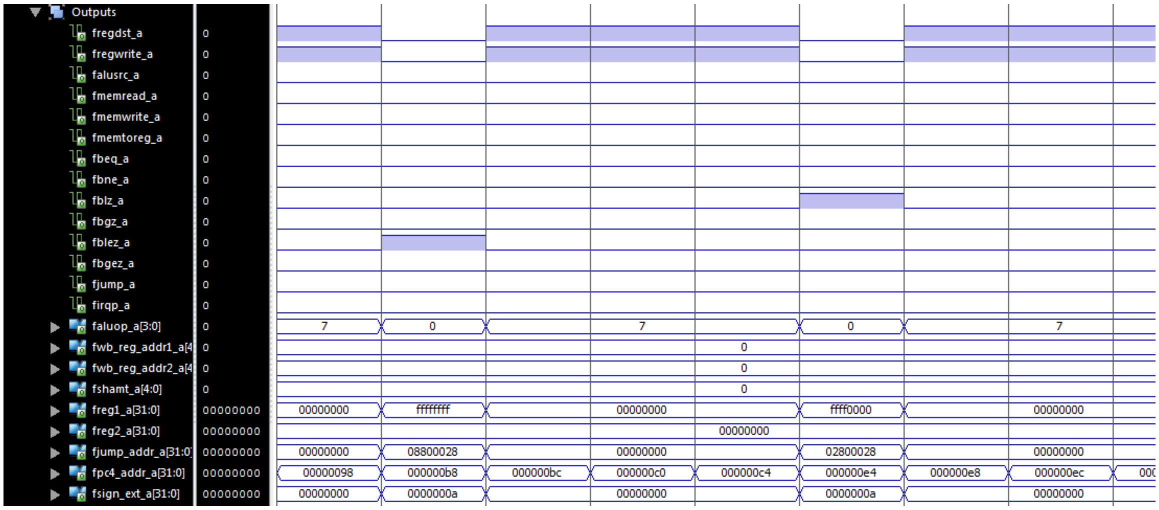

Name
Value
128 ns
130 ns
132 ns
134 ns
136 ns
138 ns
140 ns
142 ns
144 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]

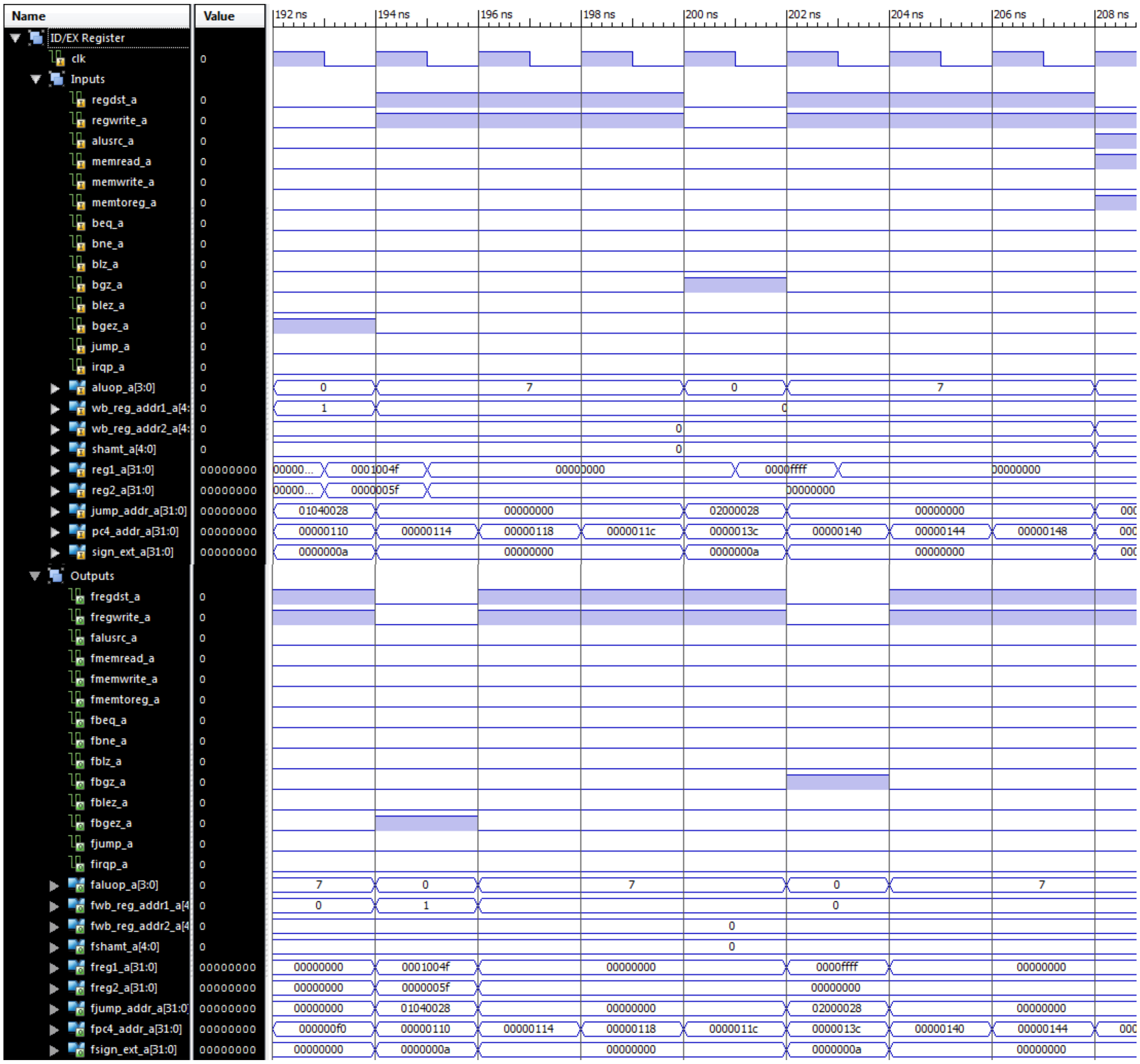

Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fwb_reg_addr1_a[4
fwb_reg_addr2_a[4
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

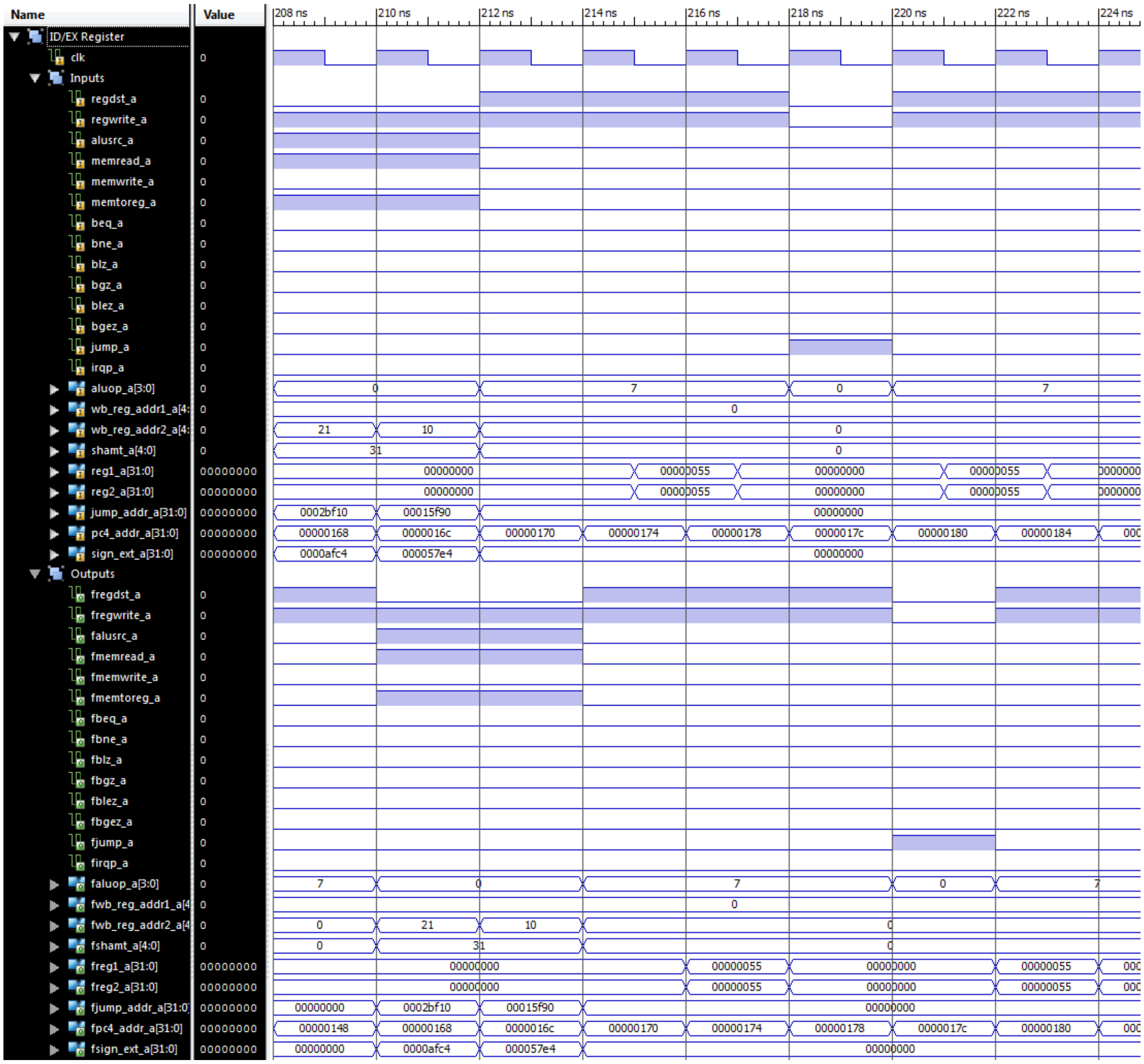

Name
Value
144 ns
146 ns
148 ns
150 ns
152 ns
154 ns
156 ns
158 ns
160 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]
Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fwb_reg_addr1_a[4
fwb_reg_addr2_a[4
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

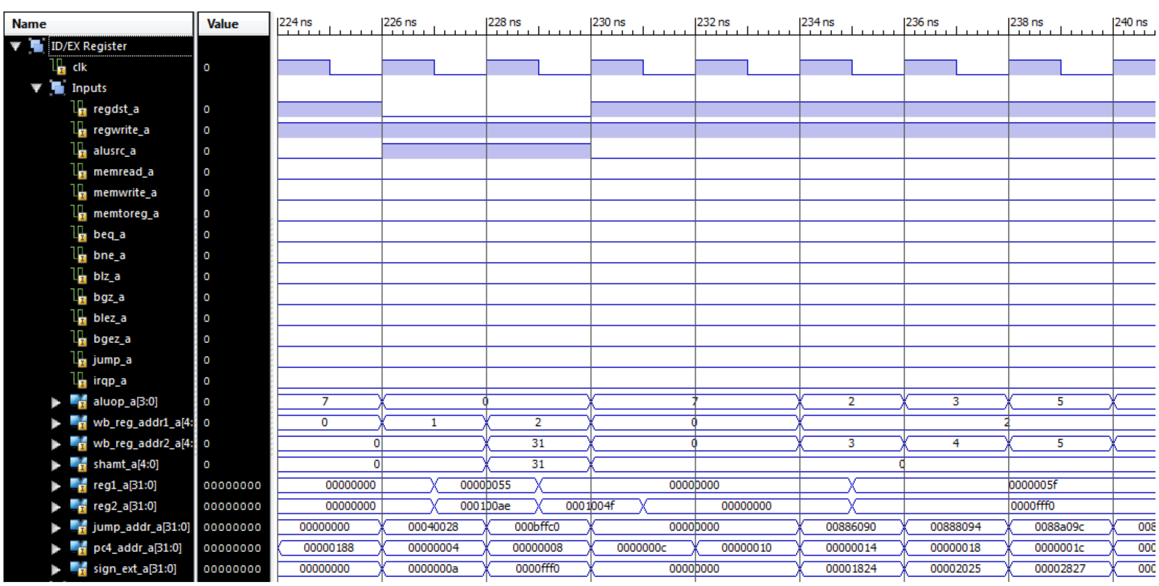


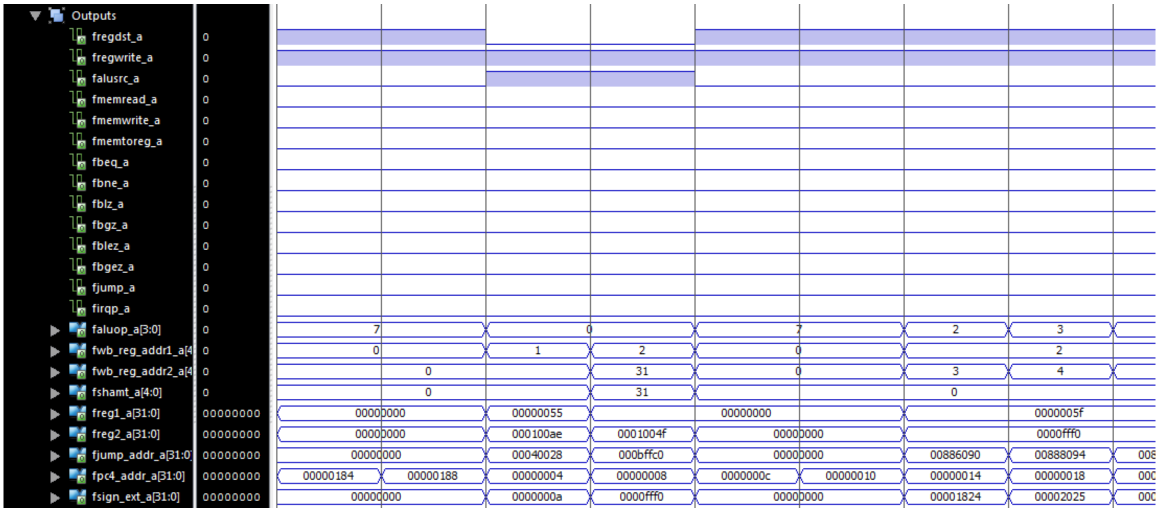


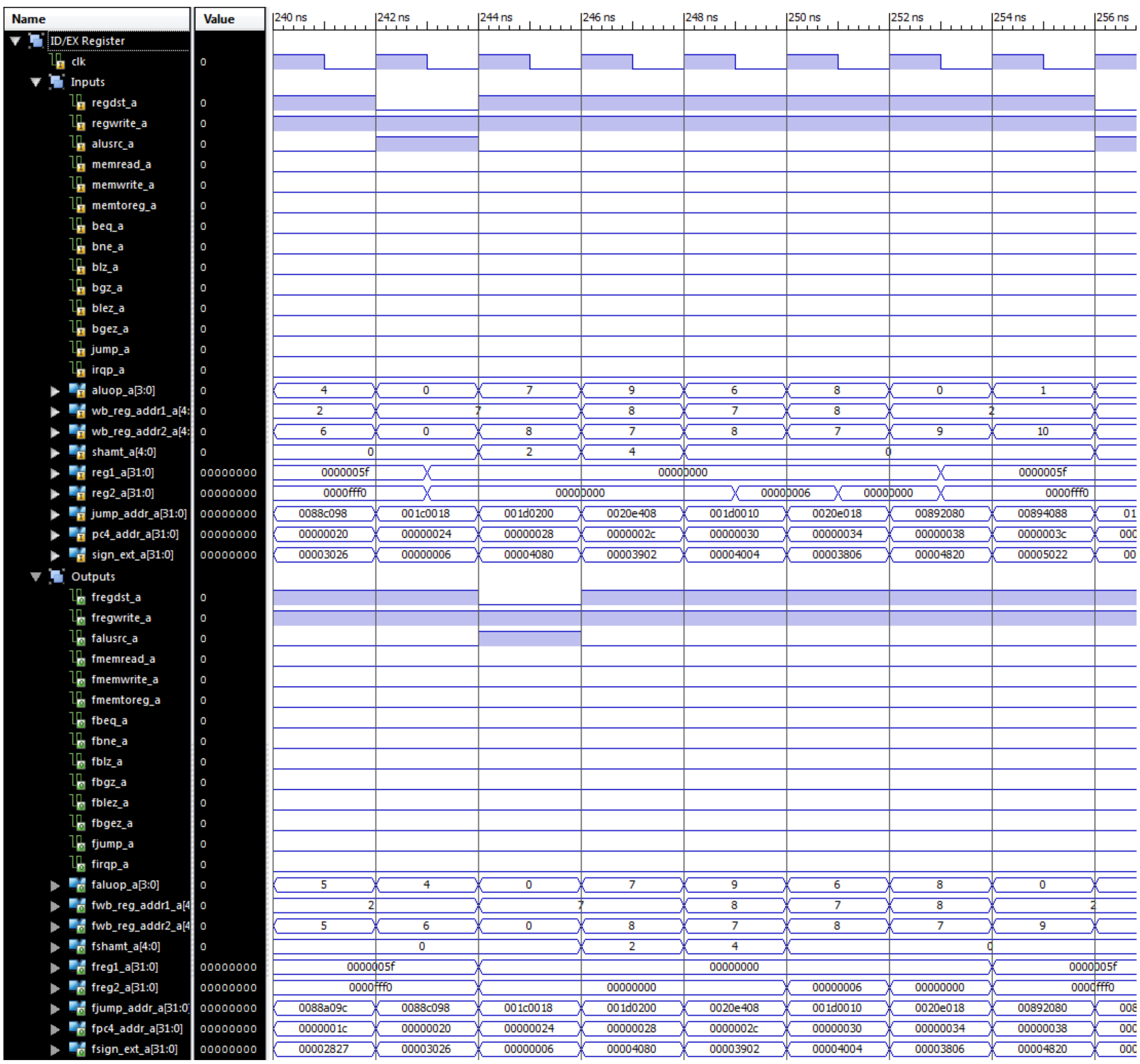


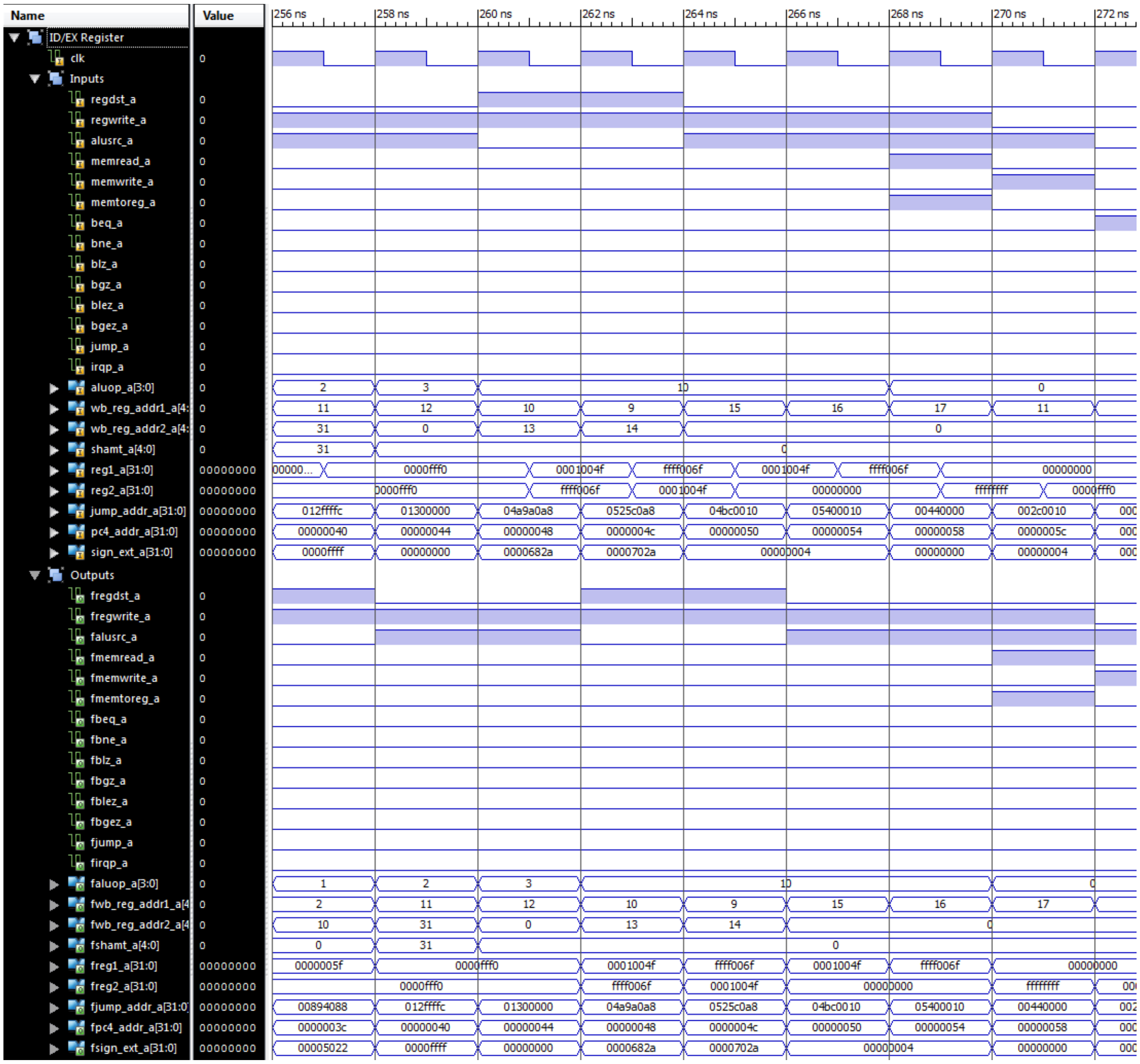


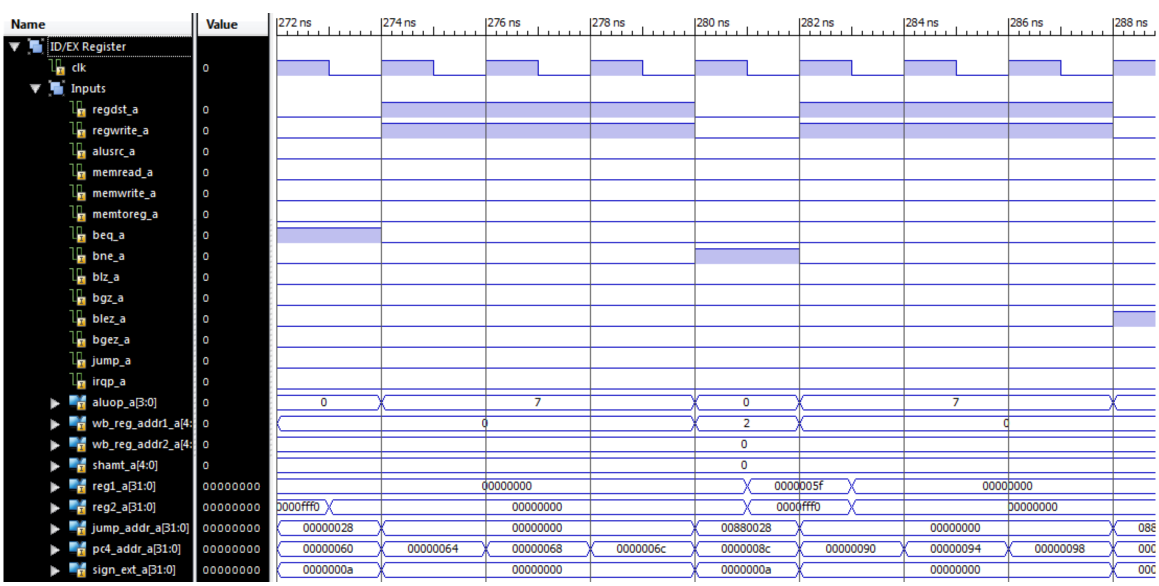

Name
Value
208 ns
210 ns
212 ns
214 ns
216 ns
218 ns
220 ns
222 ns
224 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]
Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

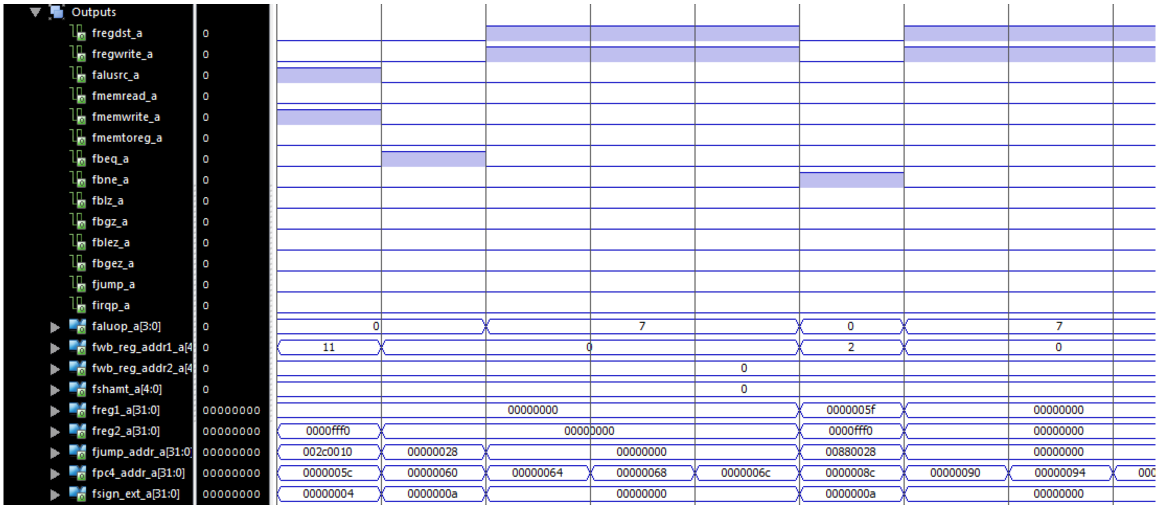

Name
Value
224 ns
226 ns
228 ns
230 ns
232 ns
234 ns
236 ns
238 ns
240 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]


Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fwb_reg_addr1_a[4
fwb_reg_addr2_a[4
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

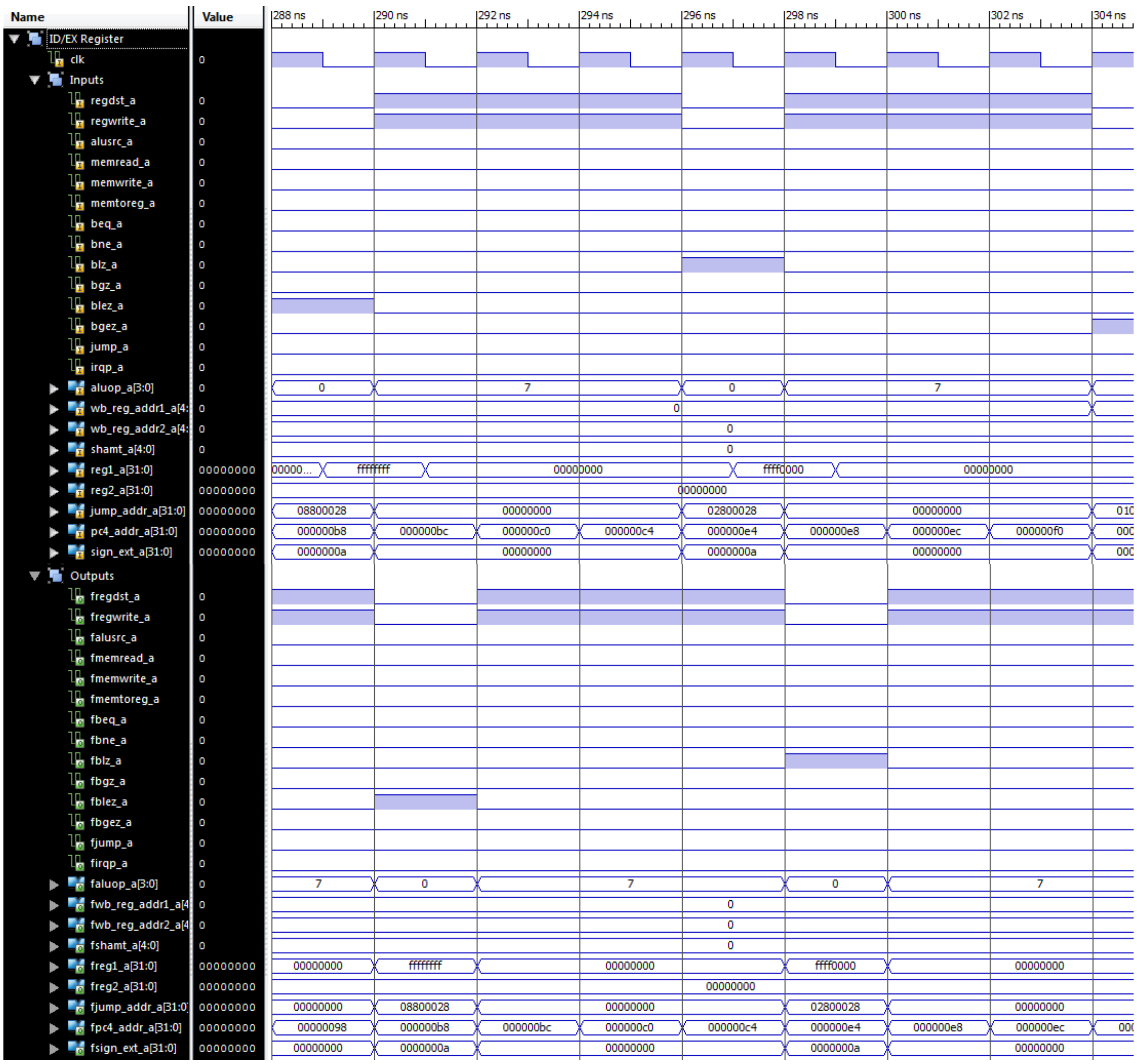

Name
Value
240 ns
242 ns
244 ns
246 ns
248 ns
250 ns
252 ns
254 ns
256 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]
Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fwb_reg_addr1_a[4
fwb_reg_addr2_a[4
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]


Name
Value
256 ns
258 ns
260 ns
262 ns
264 ns
266 ns
268 ns
270 ns
272 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]
Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fwb_reg_addr1_a[4
fwb_reg_addr2_a[4
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

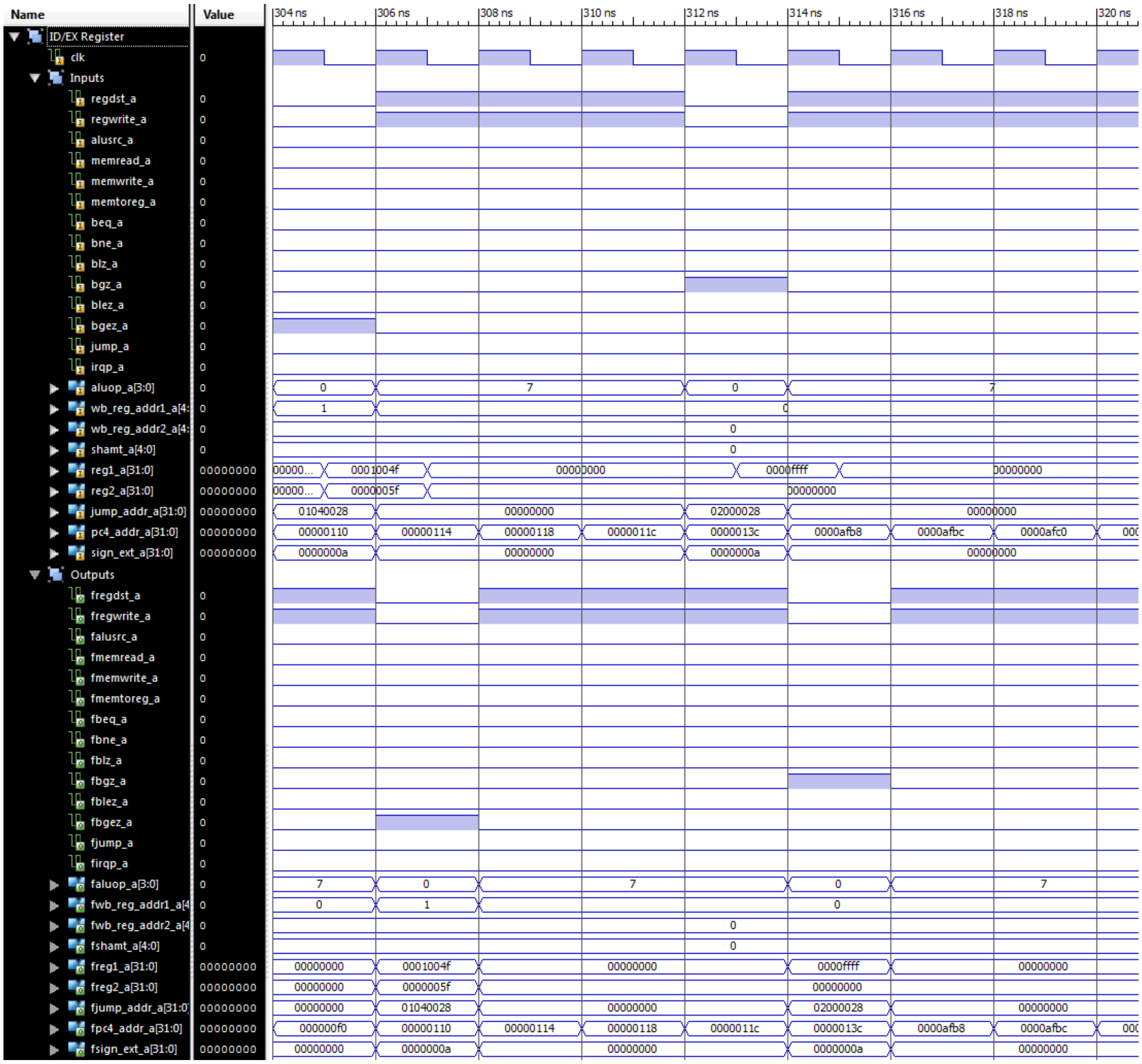

Name
Value
272 ns
274 ns
276 ns
278 ns
280 ns
282 ns
284 ns
286 ns
288 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]

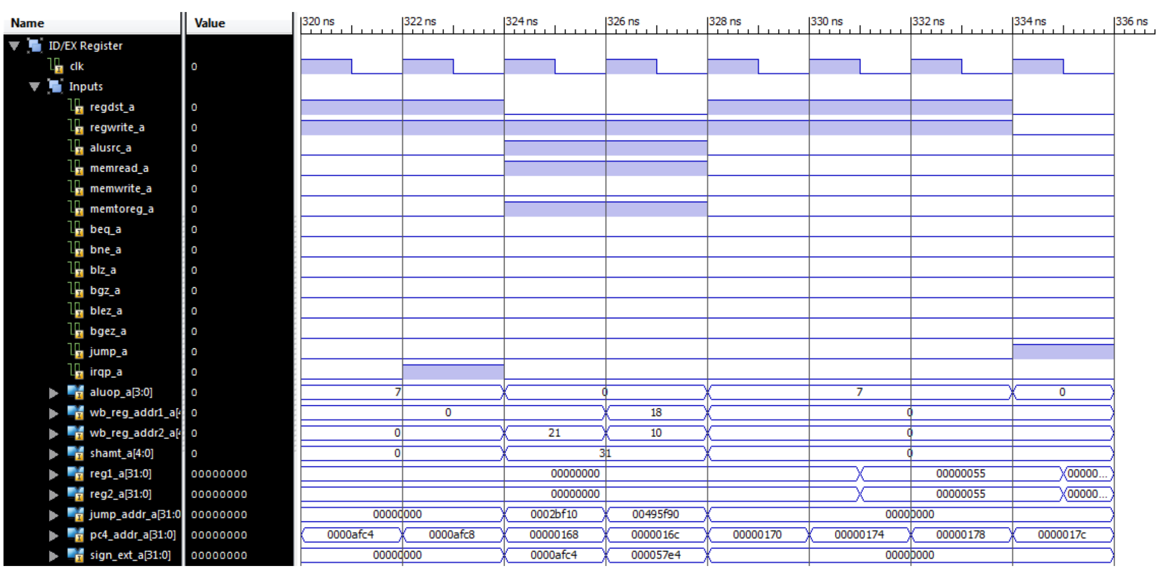


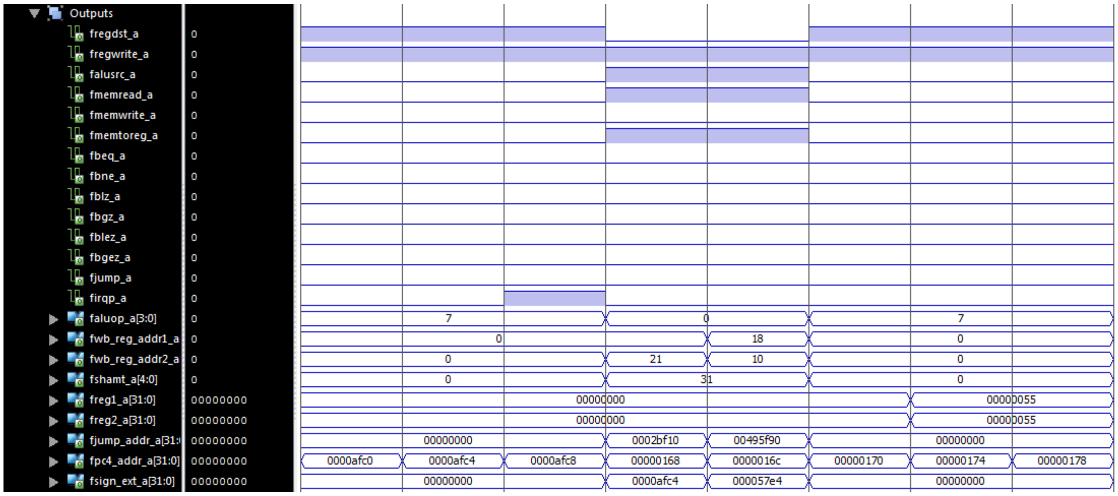
Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fwb_reg_addr1_a[4
fwb_reg_addr2_a[4
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]



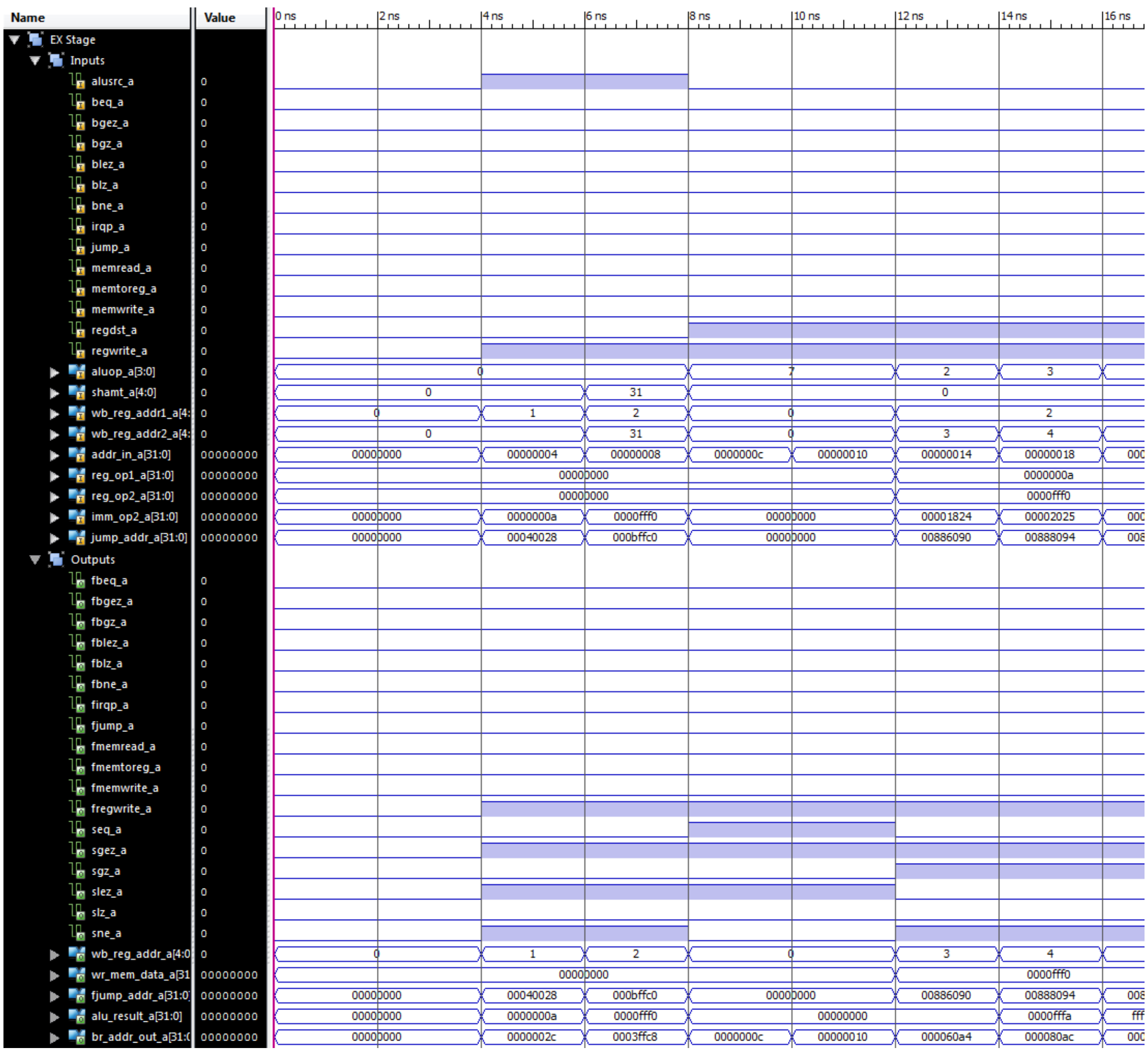
Name
Value
288 ns
290 ns
292 ns
294 ns
296 ns
298 ns
300 ns
302 ns
304 ns
ID/EX Register
clk
Inputs
regdst_a
regwrite_a
alusrc_a
memread_a
memwrite_a
memtoreg_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
irqp_a
aluop_a[3:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
shamt_a[4:0]
reg1_a[31:0]
reg2_a[31:0]
jump_addr_a[31:0]
pc4_addr_a[31:0]
sign_ext_a[31:0]
Outputs
fregdst_a
fregwrite_a
falusrc_a
fmemread_a
fmemwrite_a
fmemtoreg_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
firqp_a
faluop_a[3:0]
fwb_reg_addr1_a[4
fwb_reg_addr2_a[4
fshamt_a[4:0]
freg1_a[31:0]
freg2_a[31:0]
fjump_addr_a[31:0]
fpc4_addr_a[31:0]
fsign_ext_a[31:0]

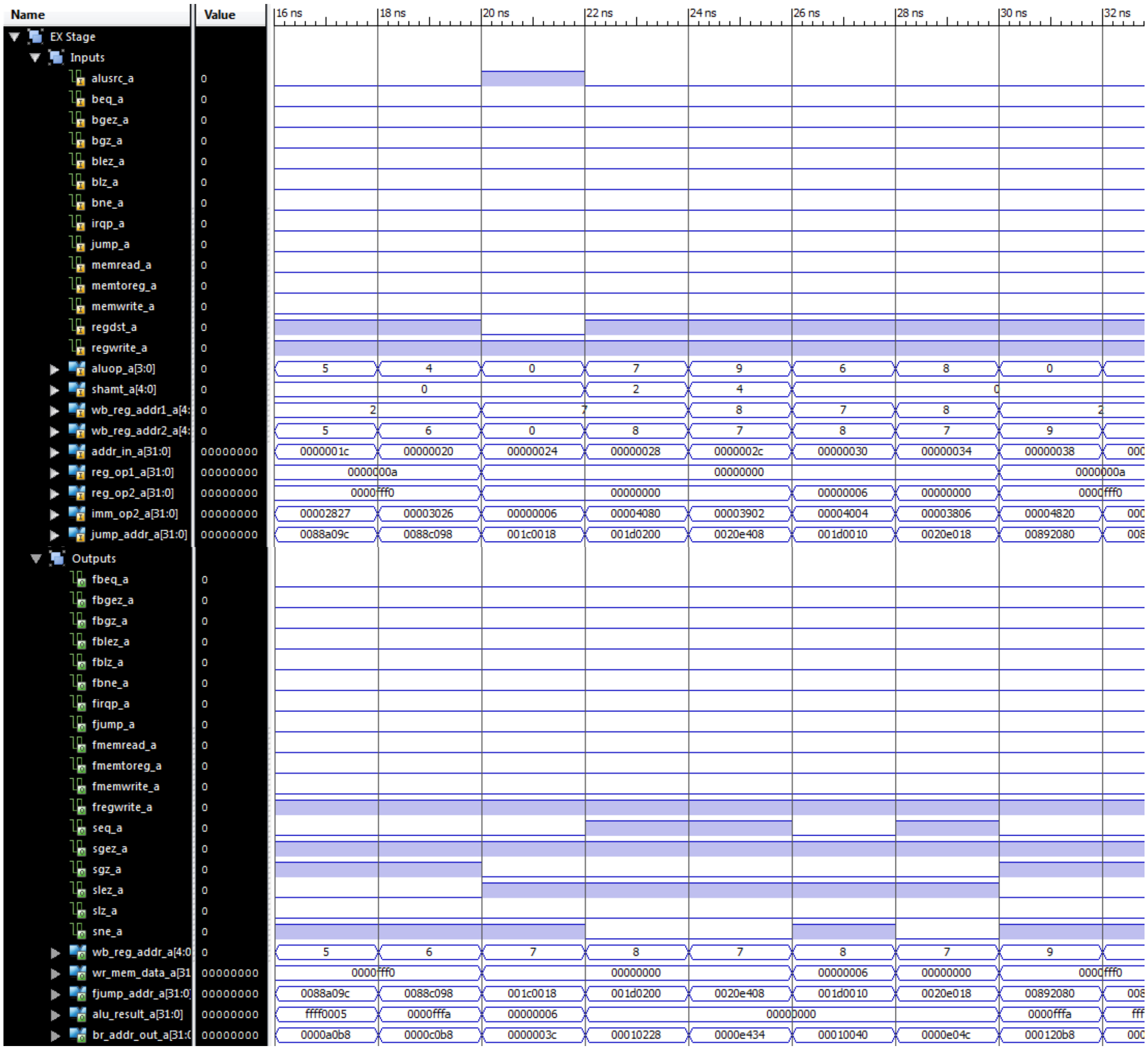


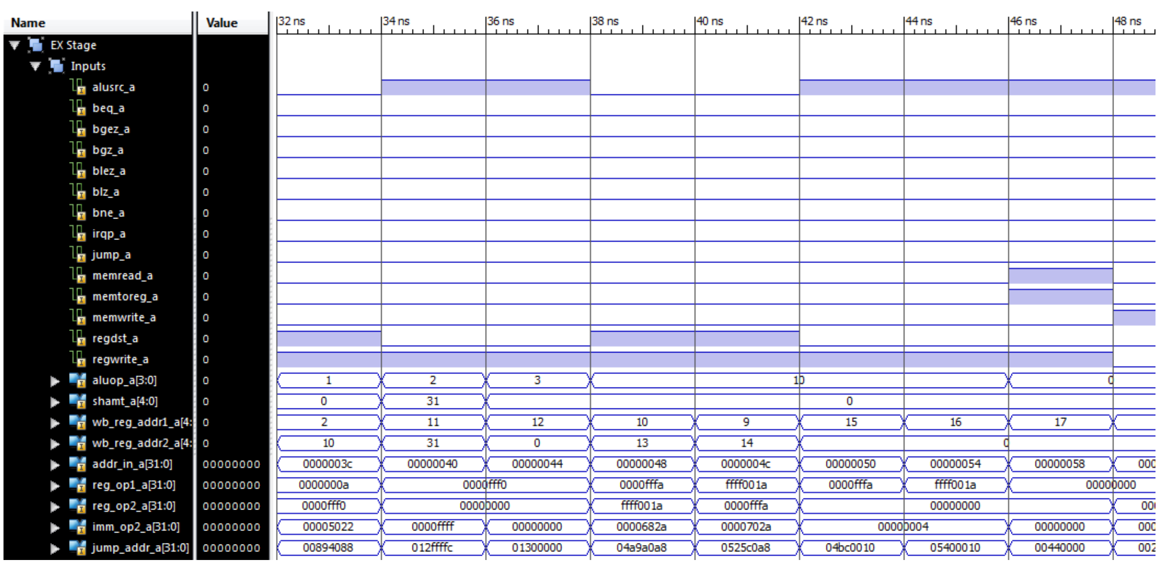


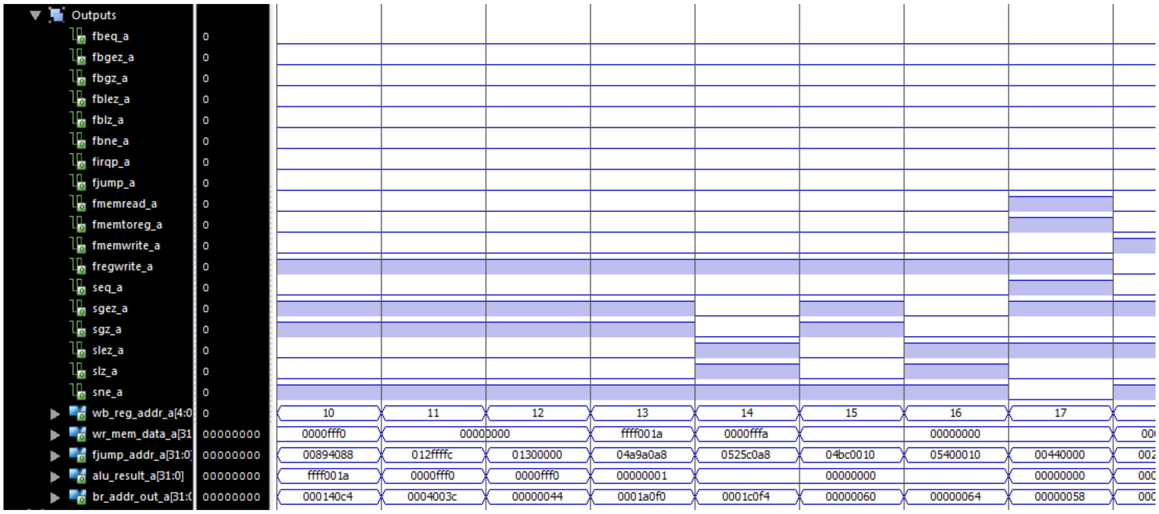


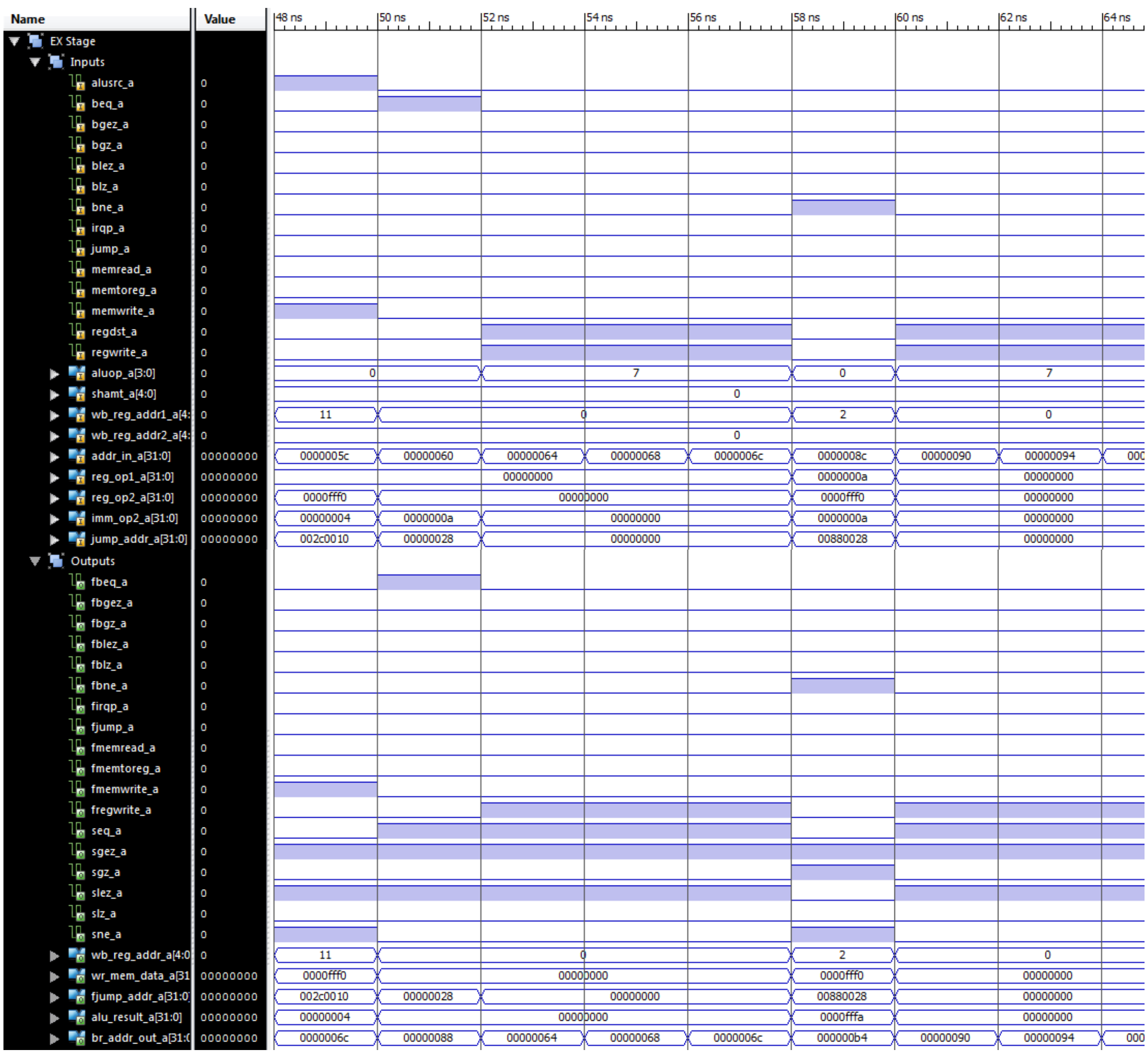
H. EX STAGE

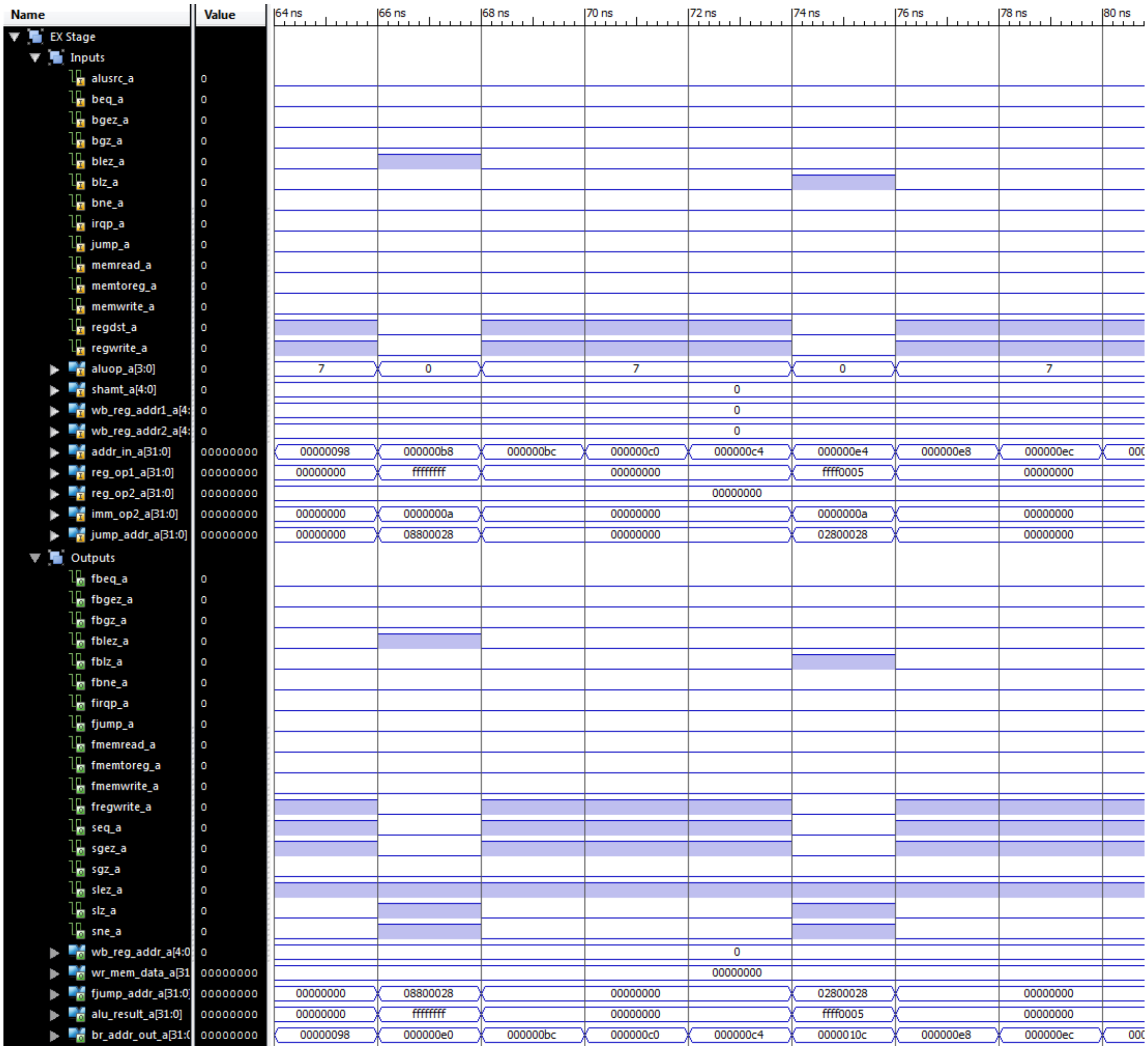


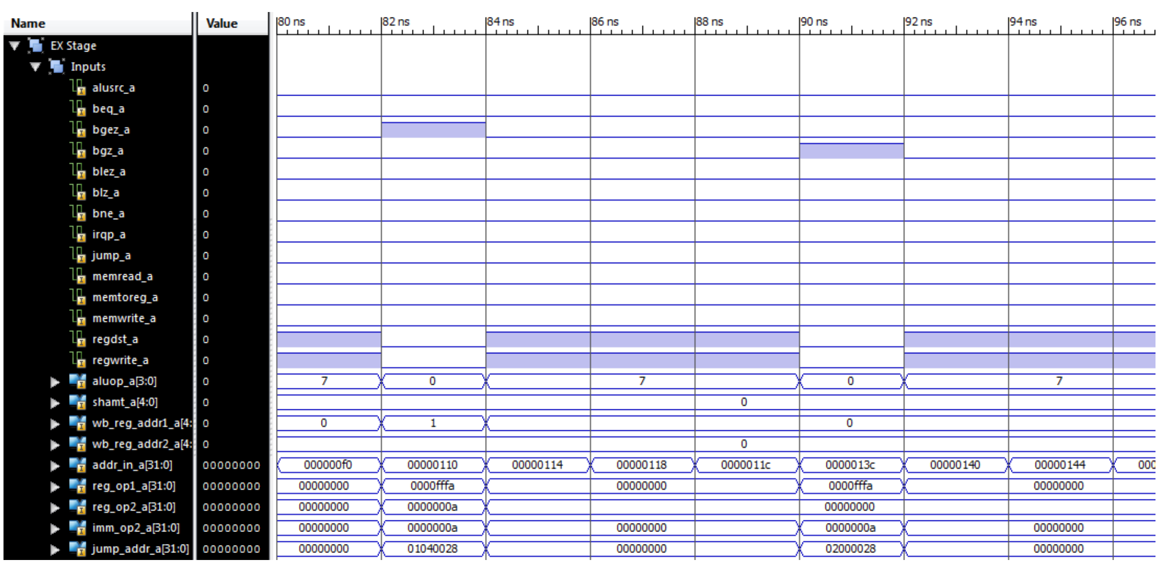


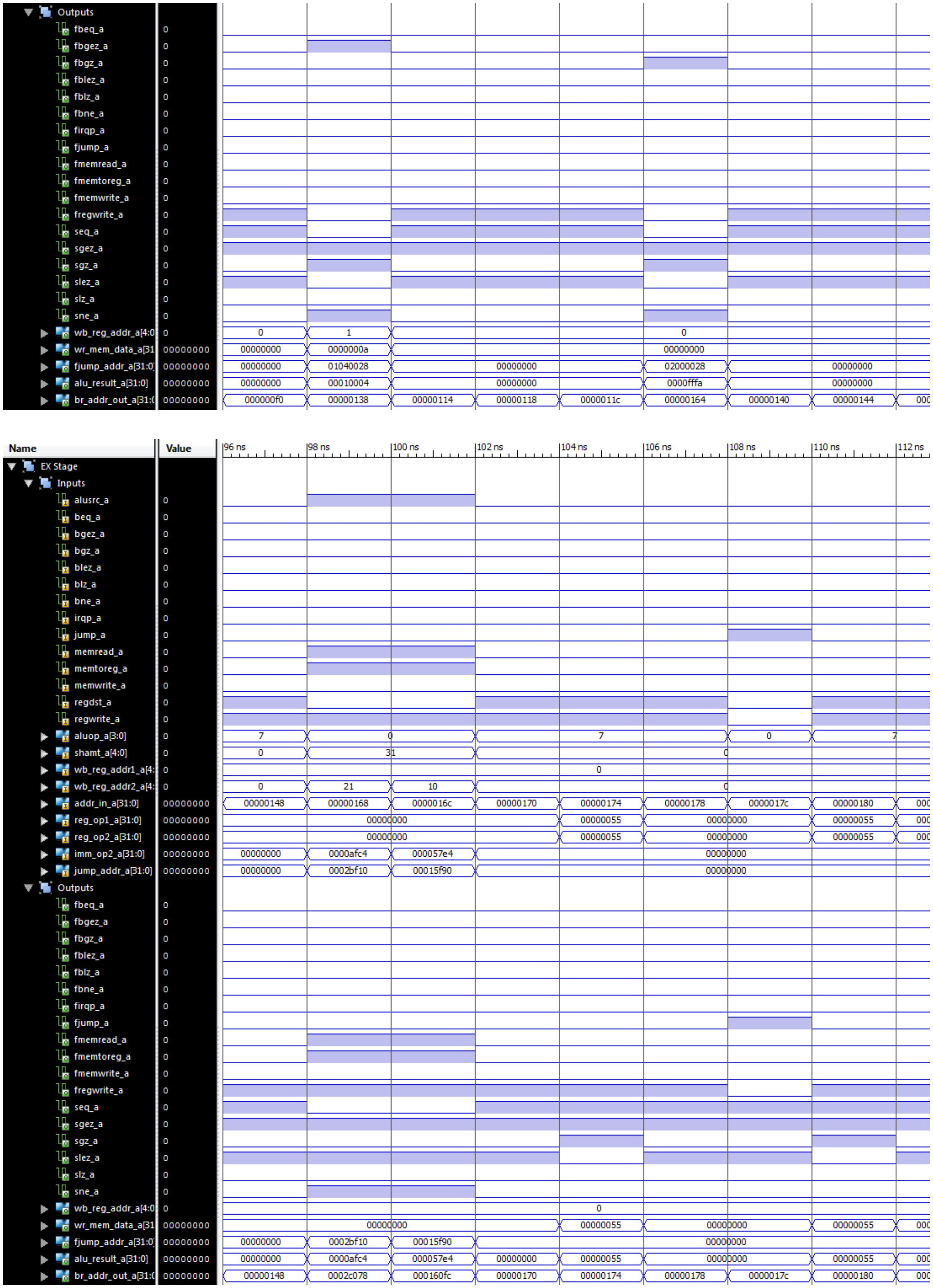


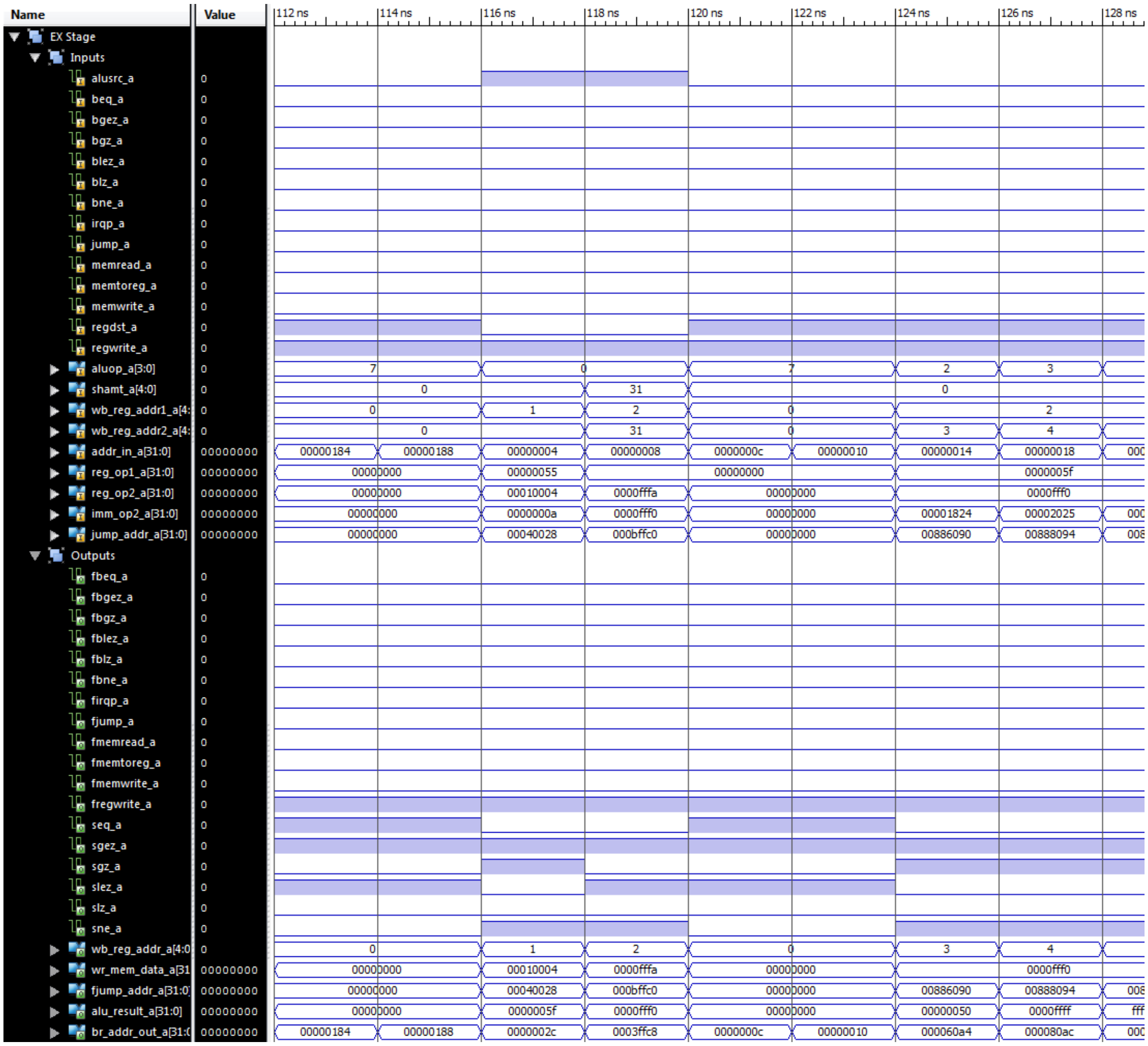


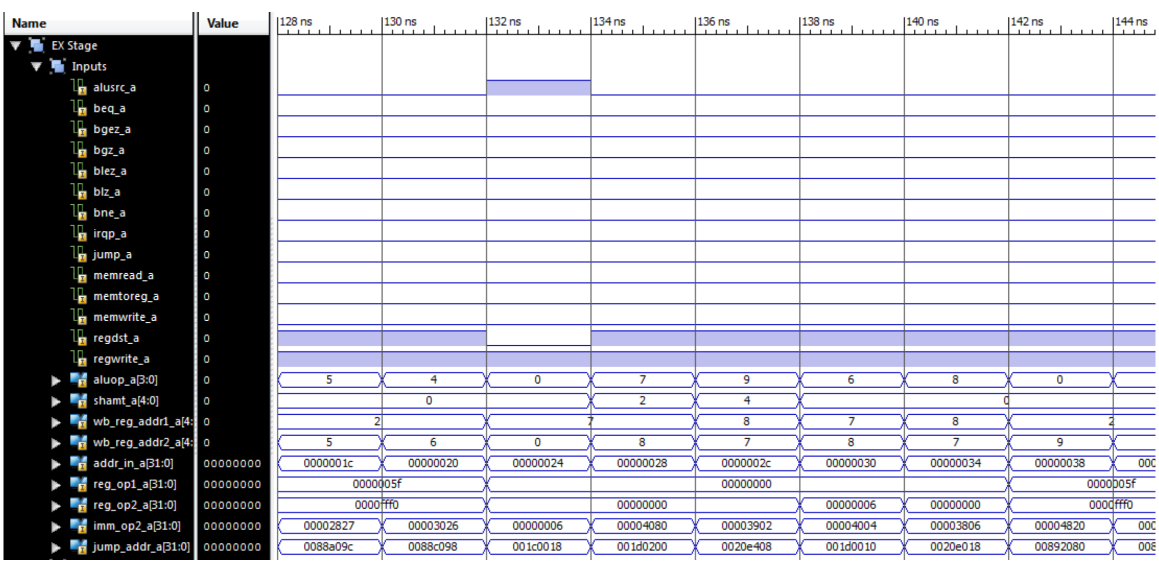


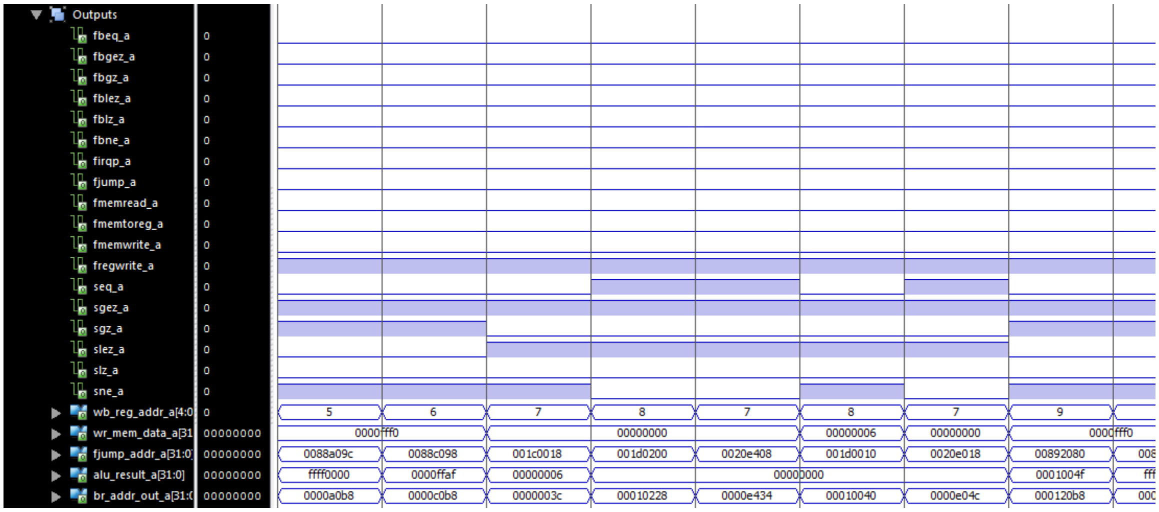

Name
Value
64 ns
66 ns
68 ns
70 ns
72 ns
74 ns
76 ns
78 ns
80 ns
EX Stage
Inputs
alusrc_a
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_a
memread_a
memtoreg_a
memwrite_a
regdst_a
regwrite_a
aluop_a[3:0]
shamt_a[4:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
addr_in_a[31:0]
reg_op1_a[31:0]
reg_op2_a[31:0]
imm_op2_a[31:0]
jump_addr_a[31:0]
Outputs
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
wb_reg_addr_a[4:0
wr_mem_data_a[31
fjump_addr_a[31:0
alu_result_a[31:0]
br_addr_out_a[31:

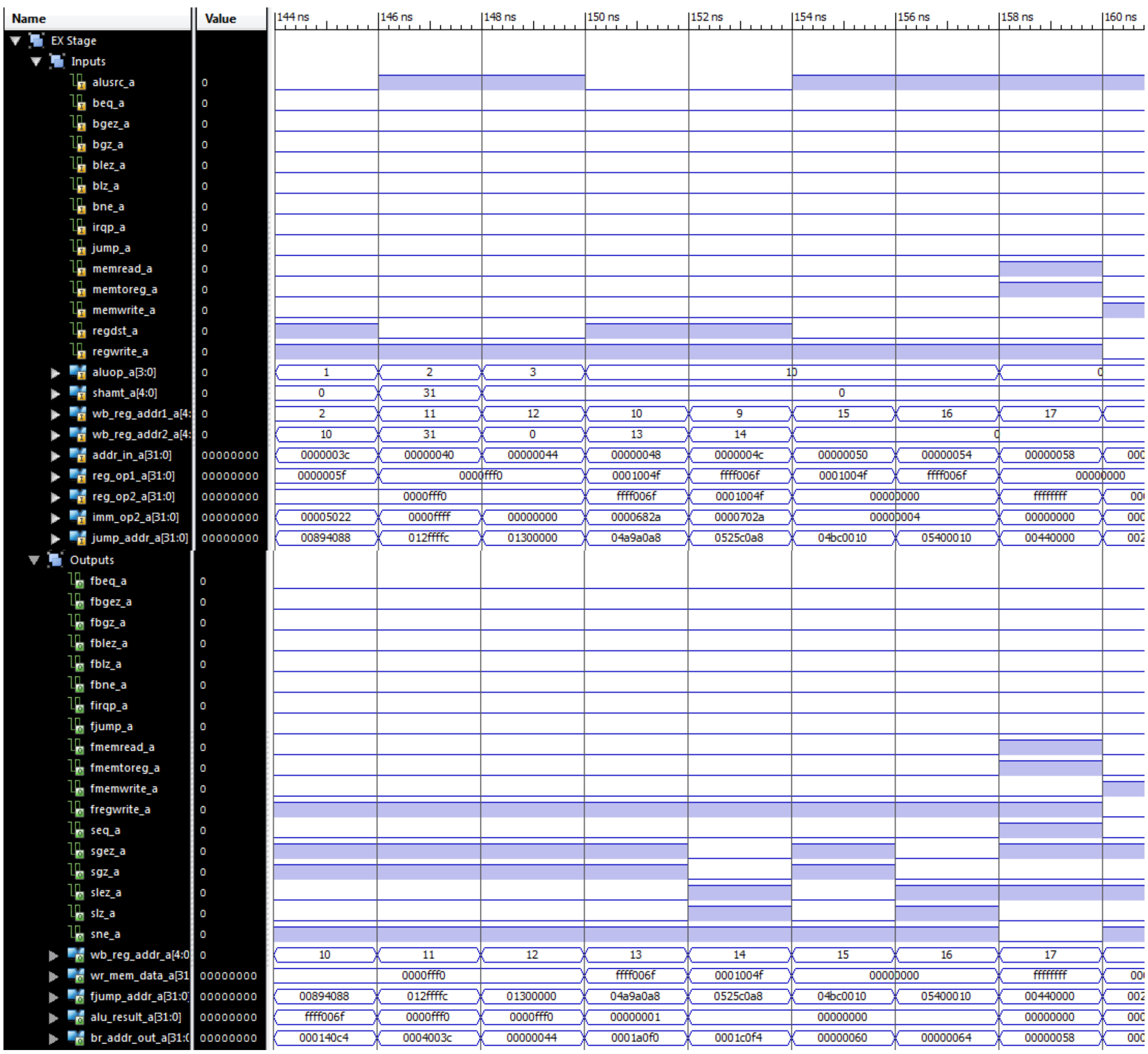

Name
Value
80 ns
82 ns
84 ns
86 ns
88 ns
90 ns
92 ns
94 ns
96 ns
EX Stage
Inputs
alusrc_a
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_a
memread_a
memtoreg_a
memwrite_a
regdst_a
regwrite_a
aluop_a[3:0]
shamt_a[4:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
addr_in_a[31:0]
reg_op1_a[31:0]
reg_op2_a[31:0]
imm_op2_a[31:0]
jump_addr_a[31:0]

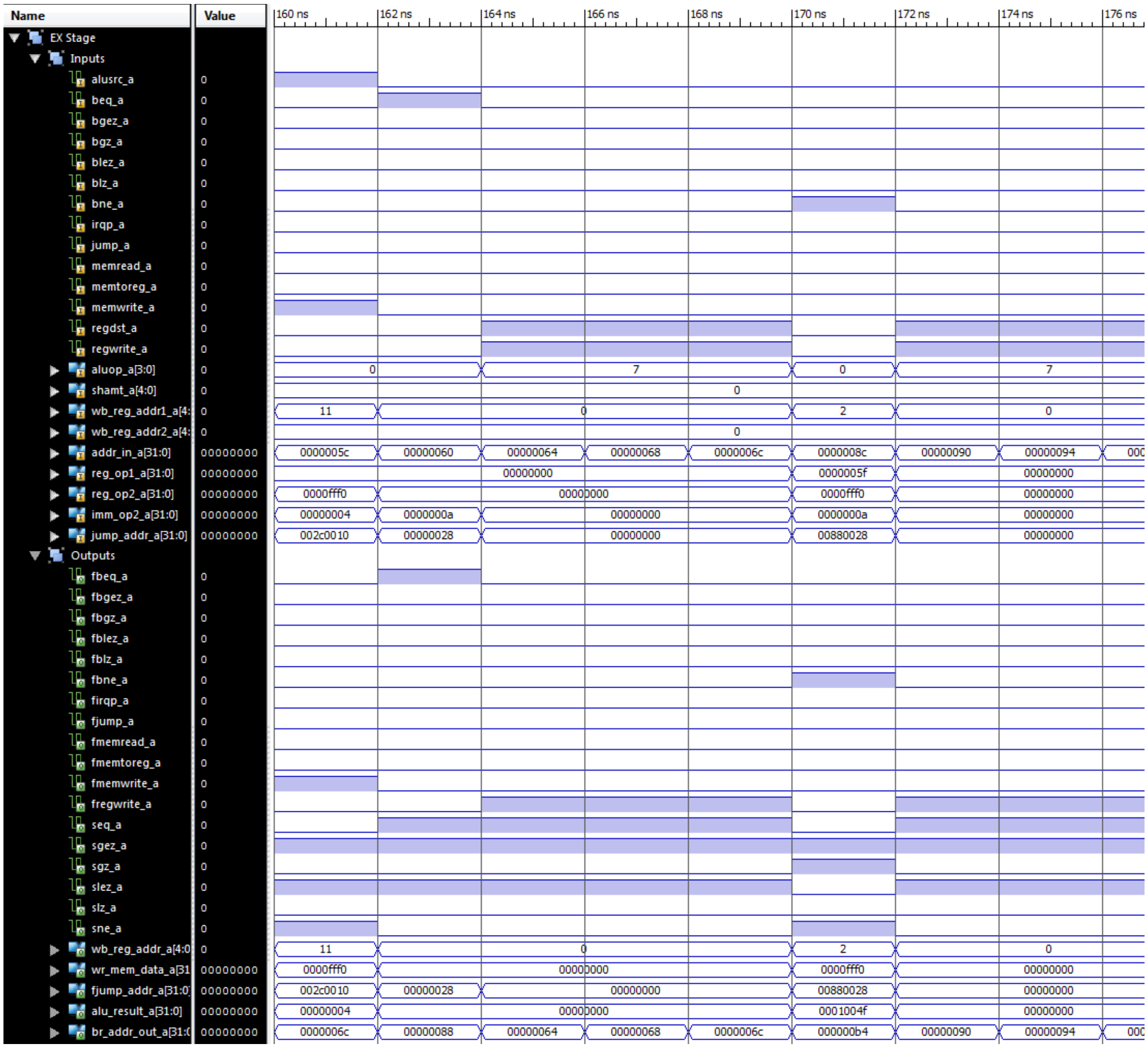


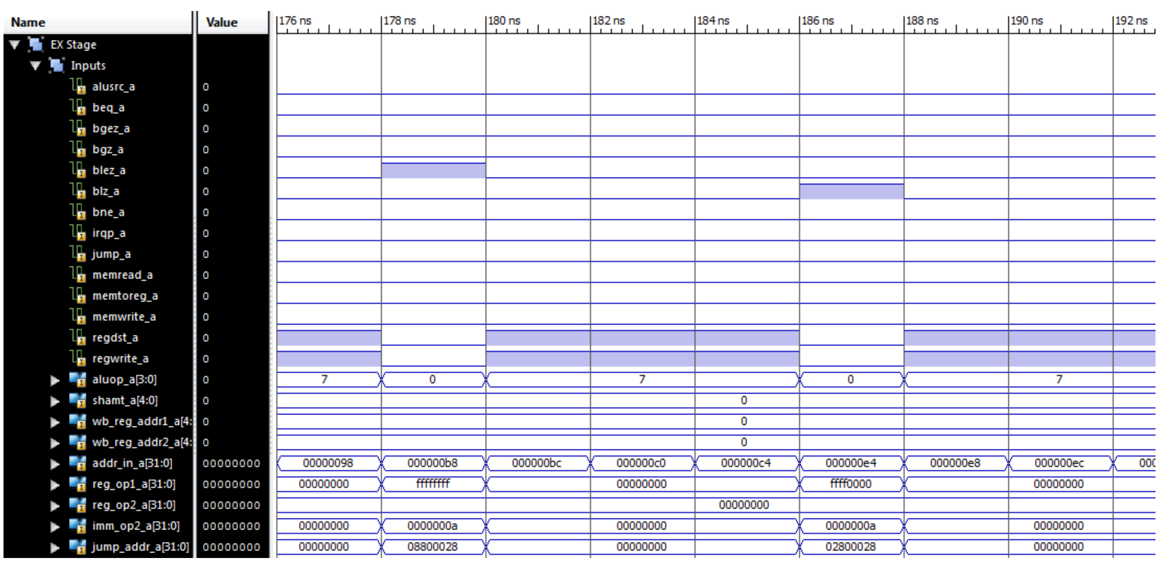

Name
Value
112 ns
114 ns
116 ns
118 ns
120 ns
122 ns
124 ns
126 ns
128 ns
EX Stage
Inputs
alusrc_a
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_a
memread_a
memtoreg_a
memwrite_a
regdst_a
regwrite_a
aluop_a[3:0]
shamt_a[4:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
addr_in_a[31:0]
reg_op1_a[31:0]
reg_op2_a[31:0]
imm_op2_a[31:0]
jump_addr_a[31:0]
Outputs
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
wb_reg_addr_a[4:0
wr_mem_data_a[31
fjump_addr_a[31:0
alu_result_a[31:0]
br_addr_out_a[31:0

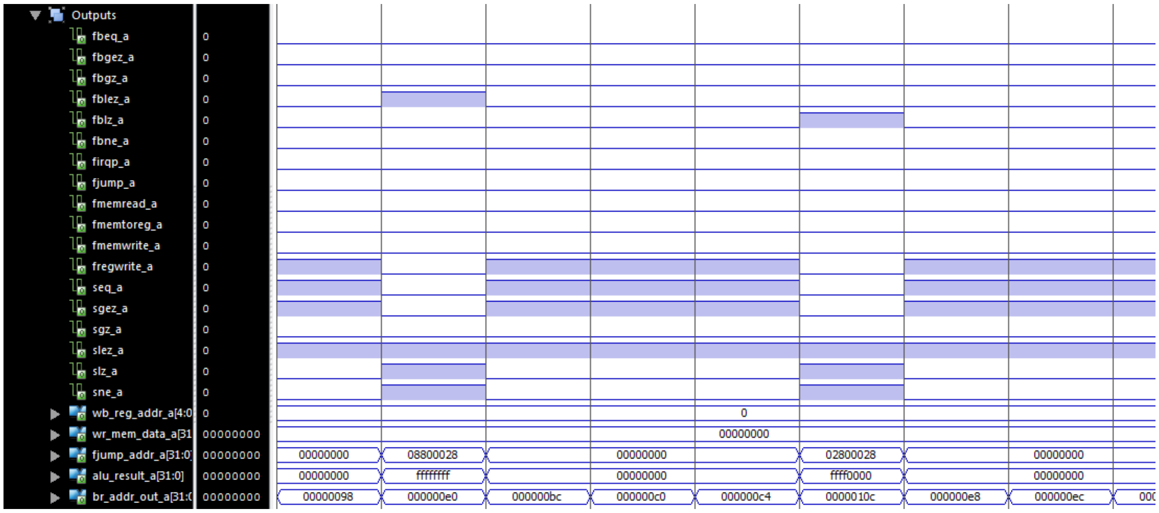

Name
Value
128 ns
130 ns
132 ns
134 ns
136 ns
138 ns
140 ns
142 ns
144 ns
EX Stage
Inputs
alusrc_a
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_a
memread_a
memtoreg_a
memwrite_a
regdst_a
regwrite_a
aluop_a[3:0]
shamt_a[4:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
addr_in_a[31:0]
reg_op1_a[31:0]
reg_op2_a[31:0]
imm_op2_a[31:0]
jump_addr_a[31:0]

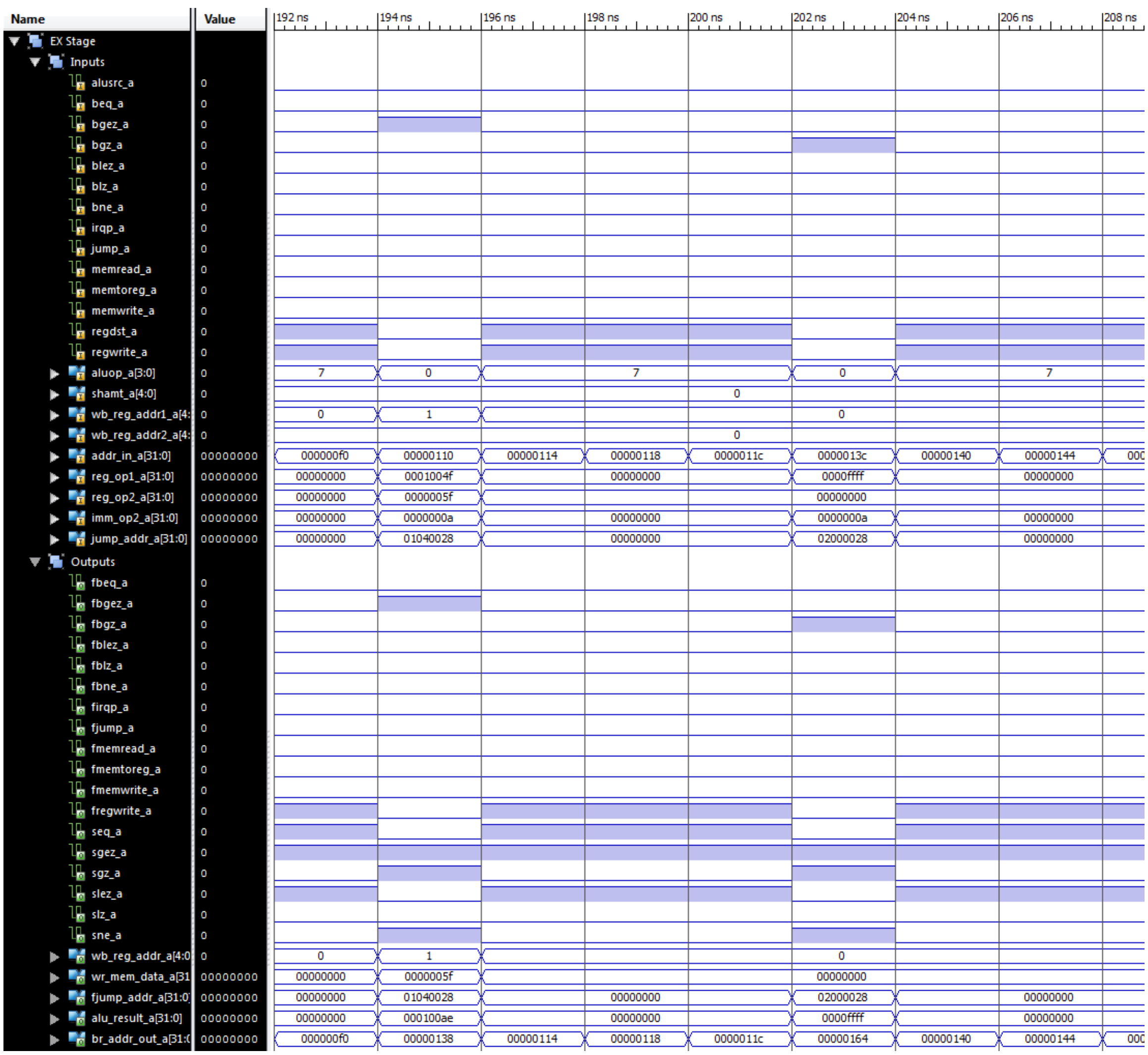


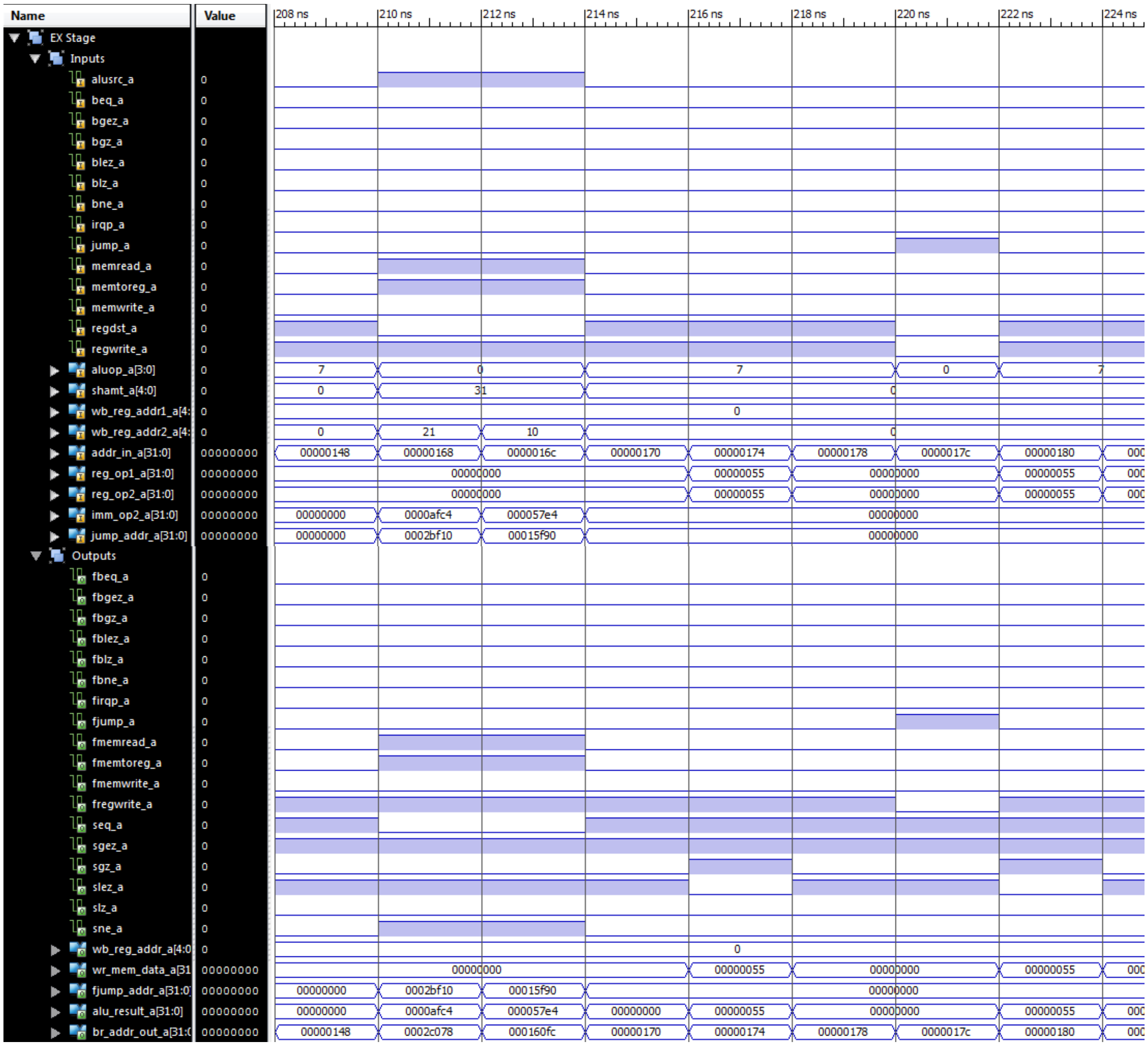


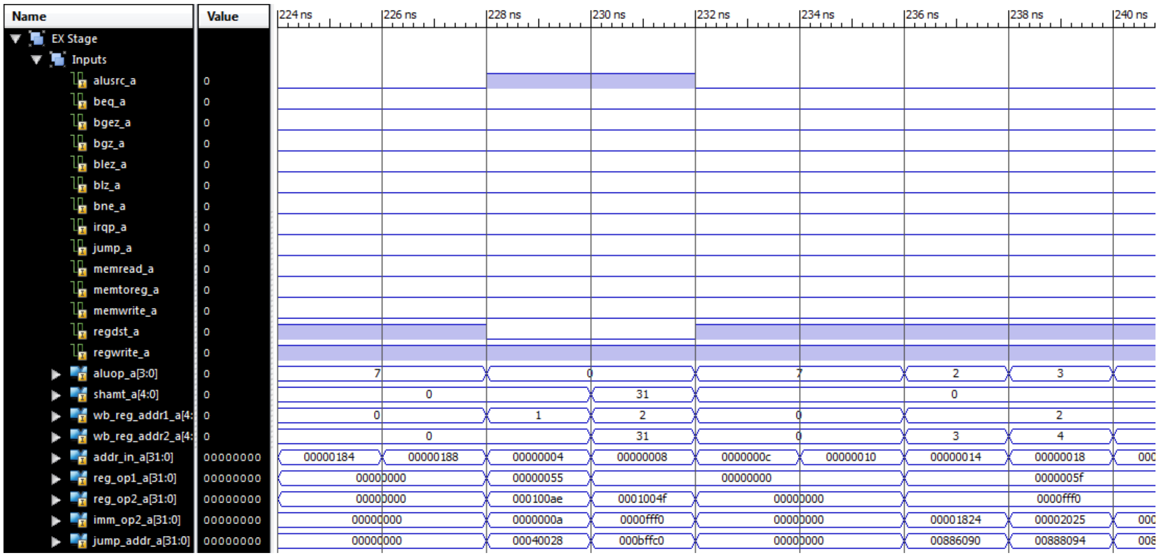


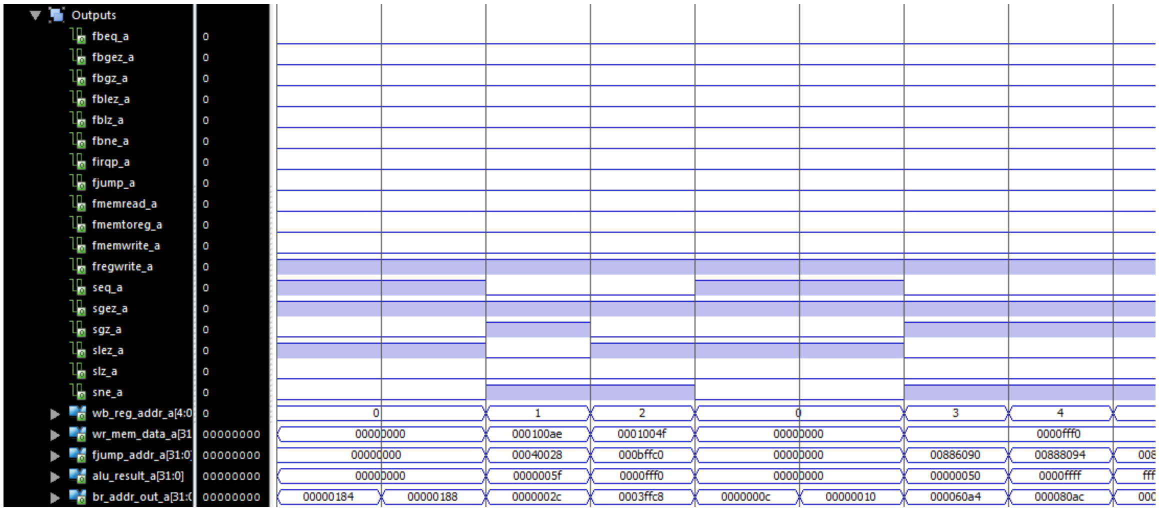


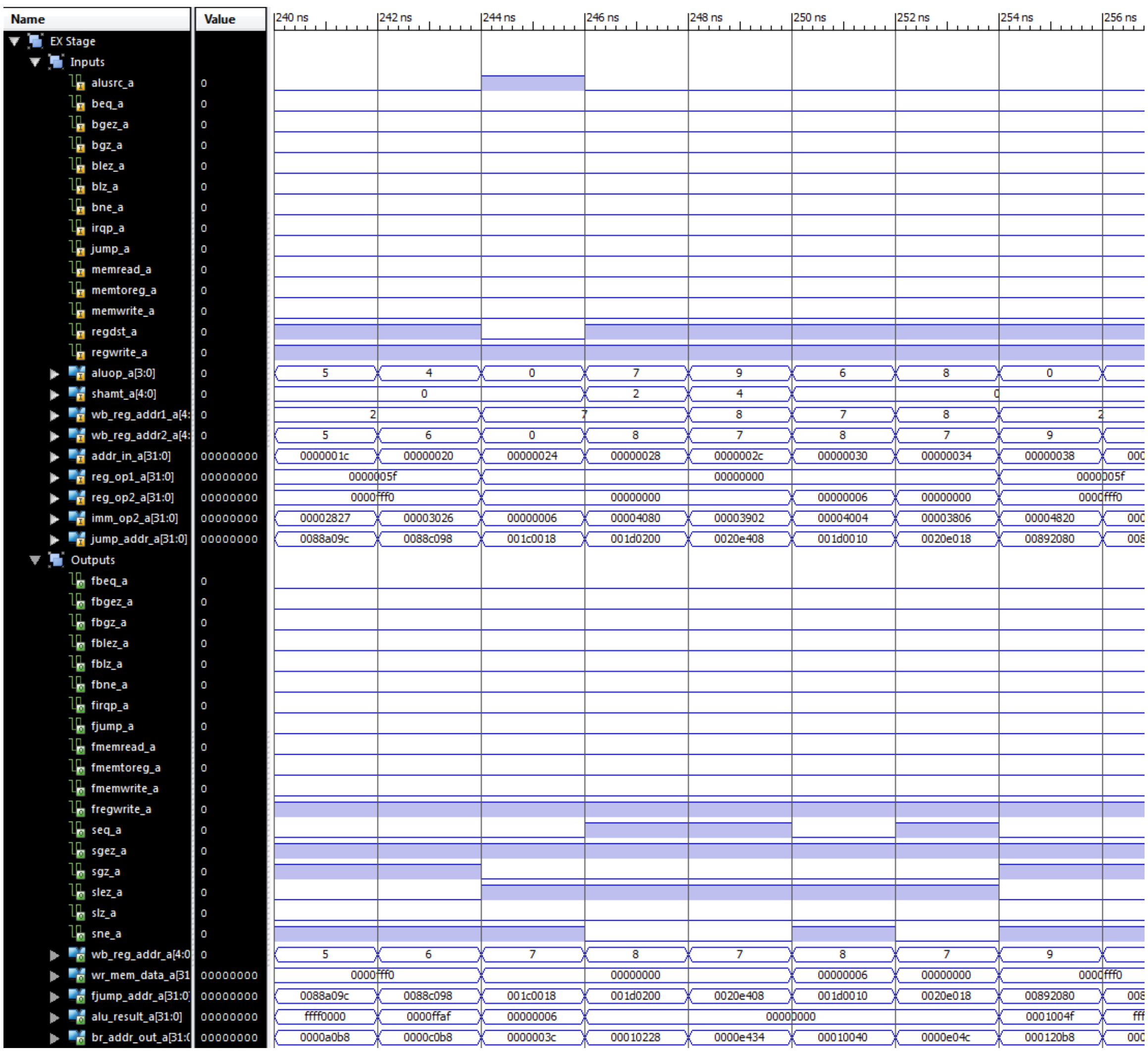

Outputs
fbeq_a 0
fbgez_a 0
fbgz_a 0
fblez_a 0
fblz_a 0
fbne_a 0
firqp_a 0
fjump_a 0
fmemread_a 0
fmemtoreg_a 0
fmemwrite_a 0
fregwrite_a 0
seq_a 0
sgez_a 0
sgz_a 0
slez_a 0
slz_a 0
sne_a 0
wb_reg_addr_a[4:0 0
0
wr_mem_data_a[31 00000000
00000000
fjump_addr_a[31:0 00000000
00000000 08800028 00000000 02800028 00000000
alu_result_a[31:0] 00000000
00000000 ffffffff 00000000 ffff0000 00000000
br_addr_out_a[31:0 00000000
00000098 000000e0 000000bc 000000c0 000000c4 0000010c 000000e8 000000ec

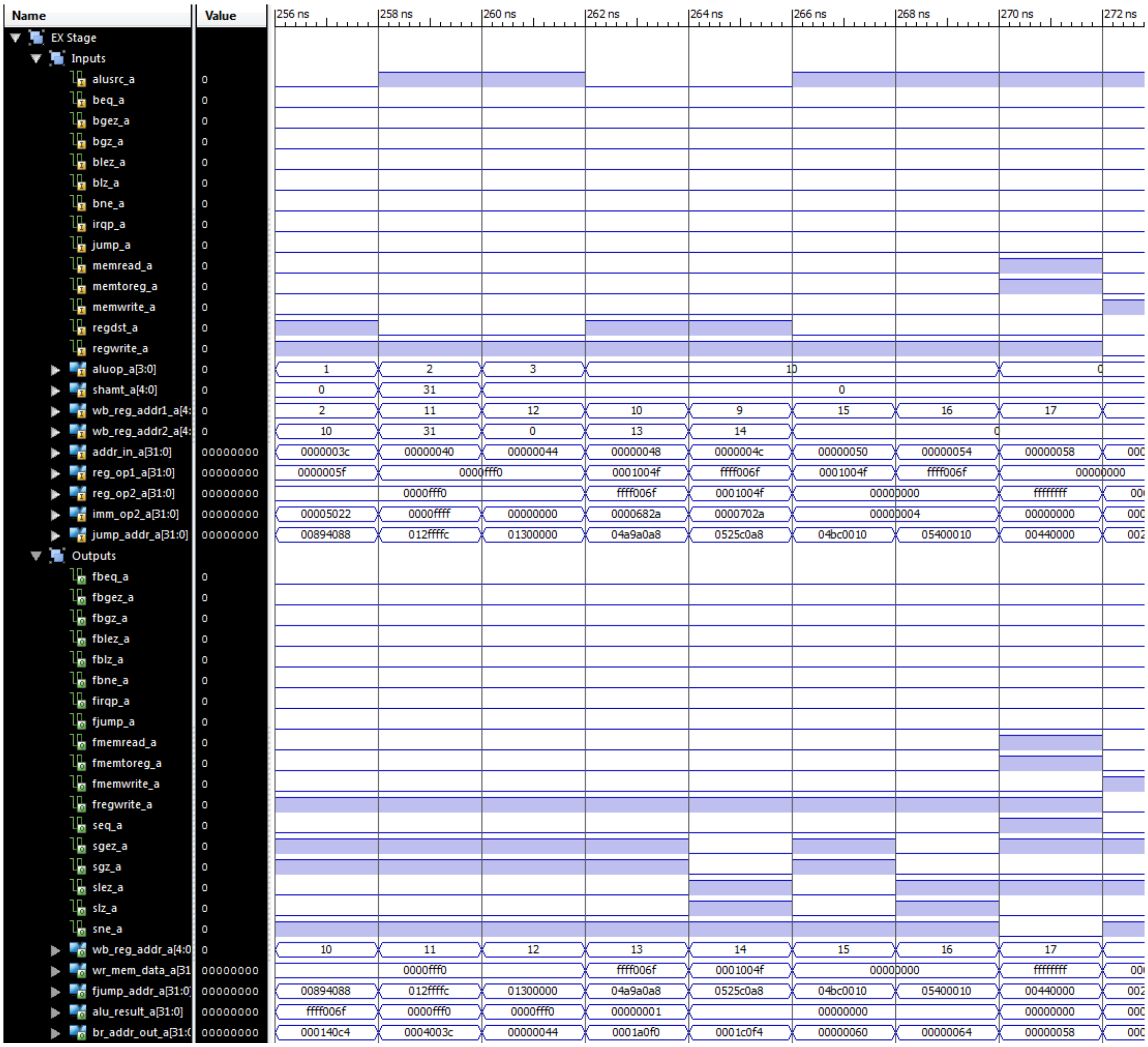

Name
Value
192 ns
194 ns
196 ns
198 ns
200 ns
202 ns
204 ns
206 ns
208 ns
EX Stage
Inputs
alusrc_a 0
beq_a 0
bgez_a 0
bgz_a 0
blez_a 0
blz_a 0
bne_a 0
irqp_a 0
jump_a 0
memread_a 0
memtoreg_a 0
memwrite_a 0
regdst_a 0
regwrite_a 0
aluop_a[3:0] 0
7 0 7 0 7
shamt_a[4:0] 0
0
wb_reg_addr1_a[4: 0
0 1 0
wb_reg_addr2_a[4: 0
0
addr_in_a[31:0] 00000000
000000f0 00000110 00000114 00000118 0000011c 0000013c 00000140 00000144
reg_op1_a[31:0] 00000000
00000000 0001004f 00000000 0000ffff 00000000
reg_op2_a[31:0] 00000000
00000000 0000005f 00000000
imm_op2_a[31:0] 00000000
00000000 0000000a 00000000 0000000a 00000000
jump_addr_a[31:0] 00000000
00000000 01040028 00000000 02000028 00000000
Outputs
fbeq_a 0
fbgez_a 0
fbgz_a 0
fblez_a 0
fblz_a 0
fbne_a 0
firqp_a 0
fjump_a 0
fmemread_a 0
fmemtoreg_a 0
fmemwrite_a 0
fregwrite_a 0
seq_a 0
sgez_a 0
sgz_a 0
slez_a 0
slz_a 0
sne_a 0
wb_reg_addr_a[4:0 0
0 1 0
wr_mem_data_a[31 00000000
00000000 0000005f 00000000
fjump_addr_a[31:0 00000000
00000000 01040028 00000000 02000028 00000000
alu_result_a[31:0] 00000000
00000000 000100ae 00000000 0000ffff 00000000
br_addr_out_a[31:0 00000000
000000f0 00000138 00000114 00000118 0000011c 00000164 00000140 00000144

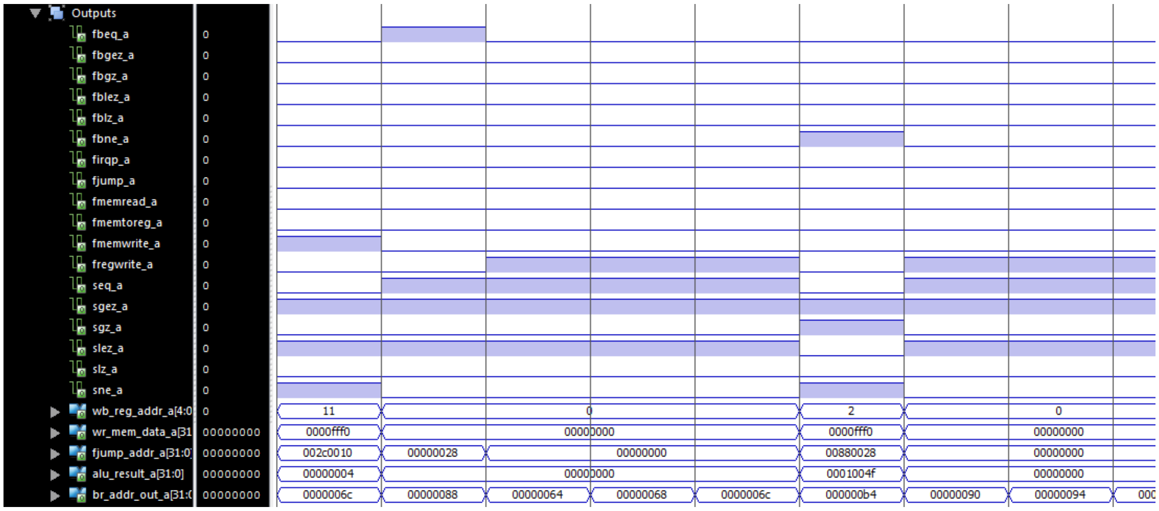


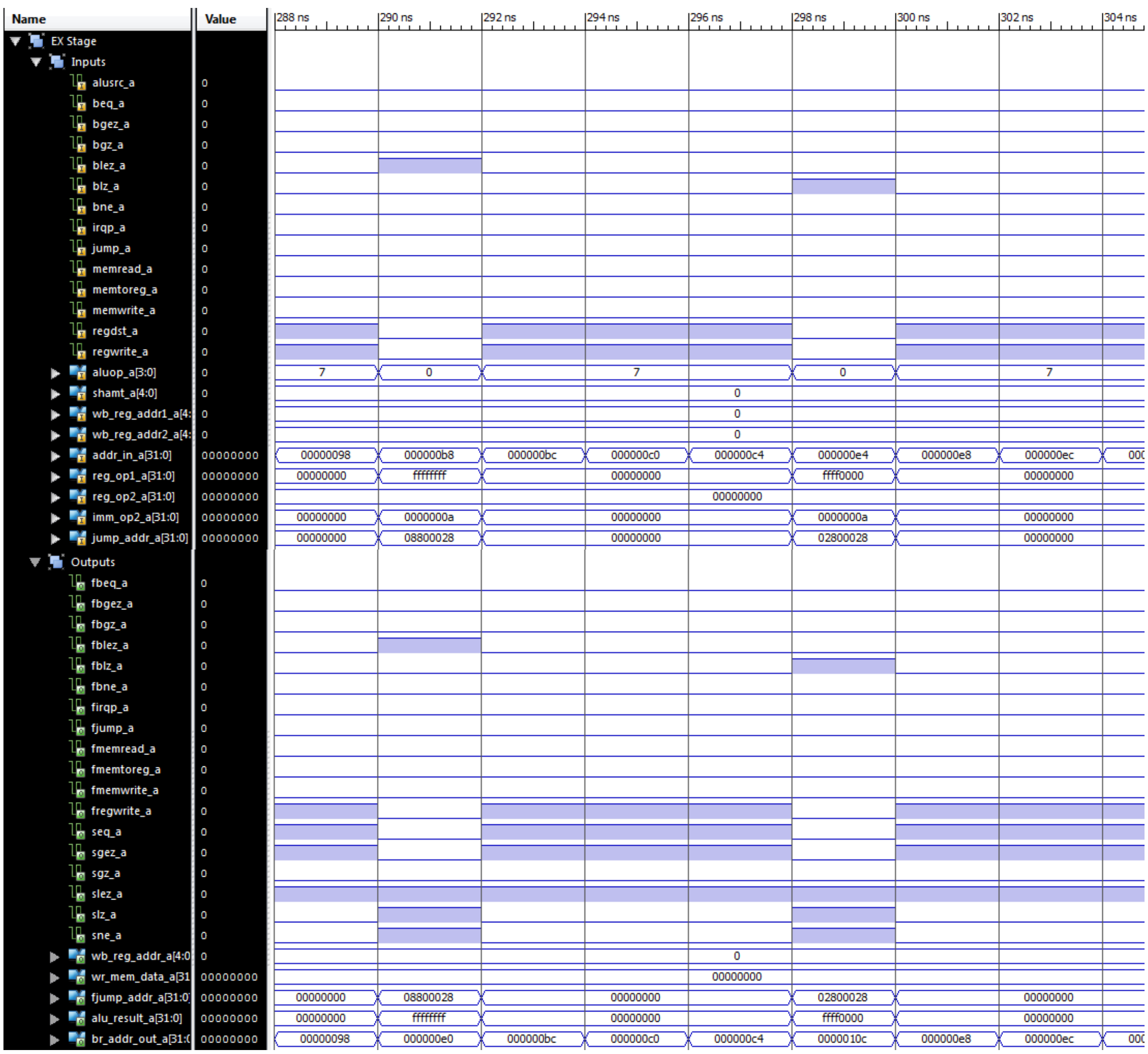


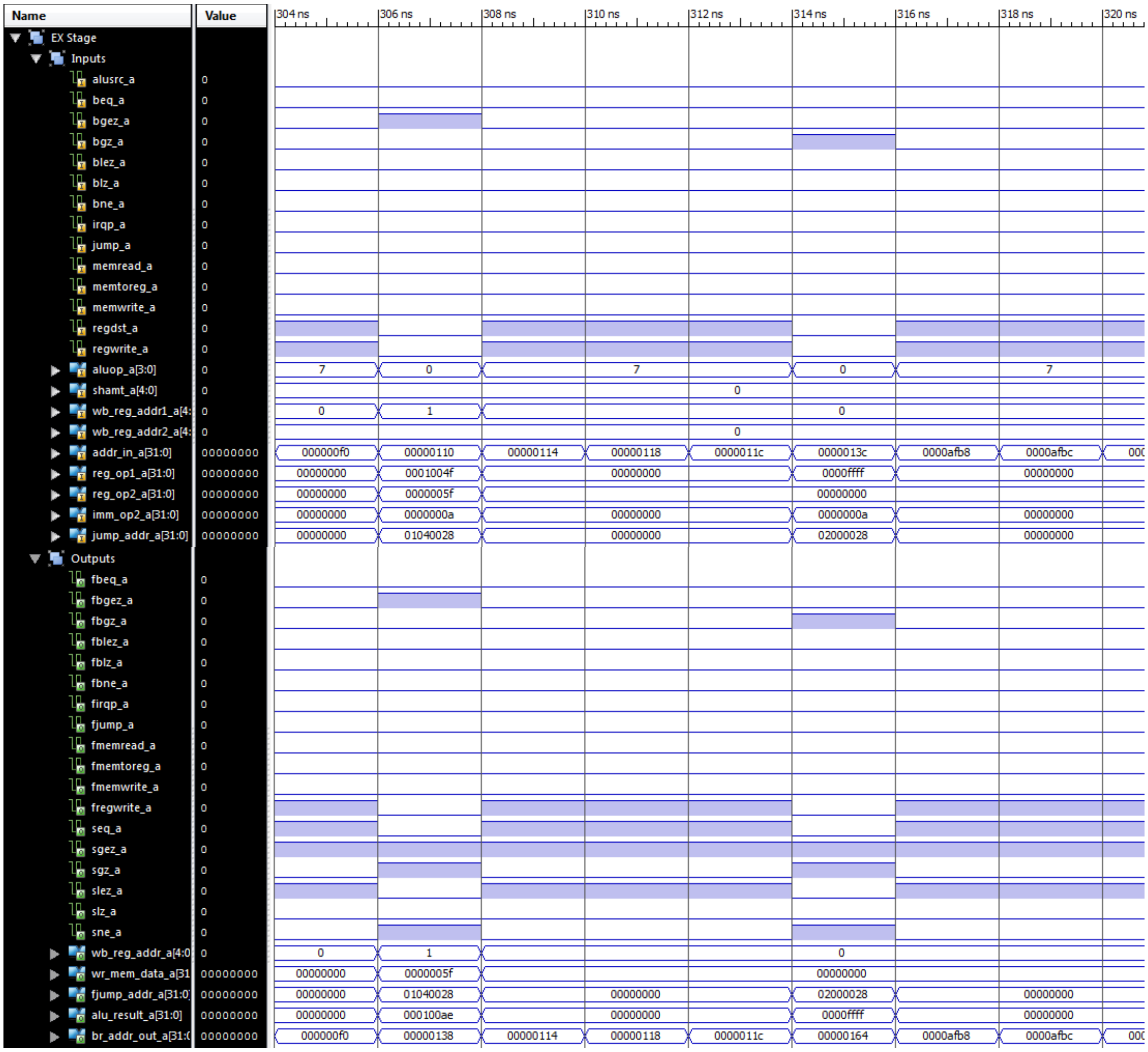

Outputs
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
wb_reg_addr_a[4:0]
wr_mem_data_a[31
fjump_addr_a[31:0]
alu_result_a[31:0]
br_addr_out_a[31:0

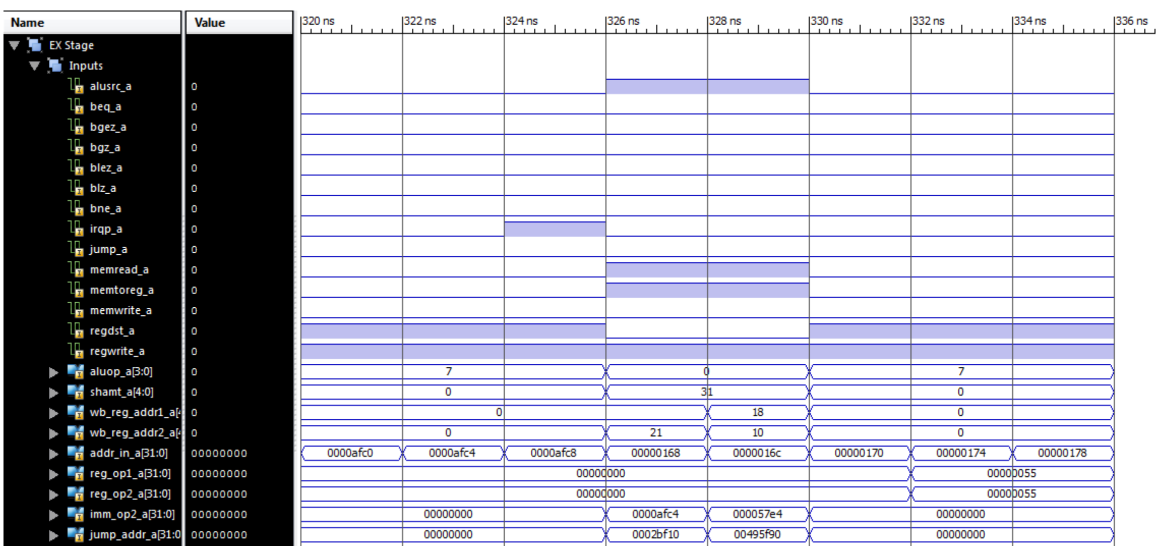

Name
Value
240 ns
242 ns
244 ns
246 ns
248 ns
250 ns
252 ns
254 ns
256 ns
EX Stage
Inputs
alusrc_a
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_a
memread_a
memtoreg_a
memwrite_a
regdst_a
regwrite_a
aluop_a[3:0]
shamt_a[4:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
addr_in_a[31:0]
reg_op1_a[31:0]
reg_op2_a[31:0]
imm_op2_a[31:0]
jump_addr_a[31:0]
Outputs
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
wb_reg_addr_a[4:0]
wr_mem_data_a[31
fjump_addr_a[31:0]
alu_result_a[31:0]
br_addr_out_a[31:0

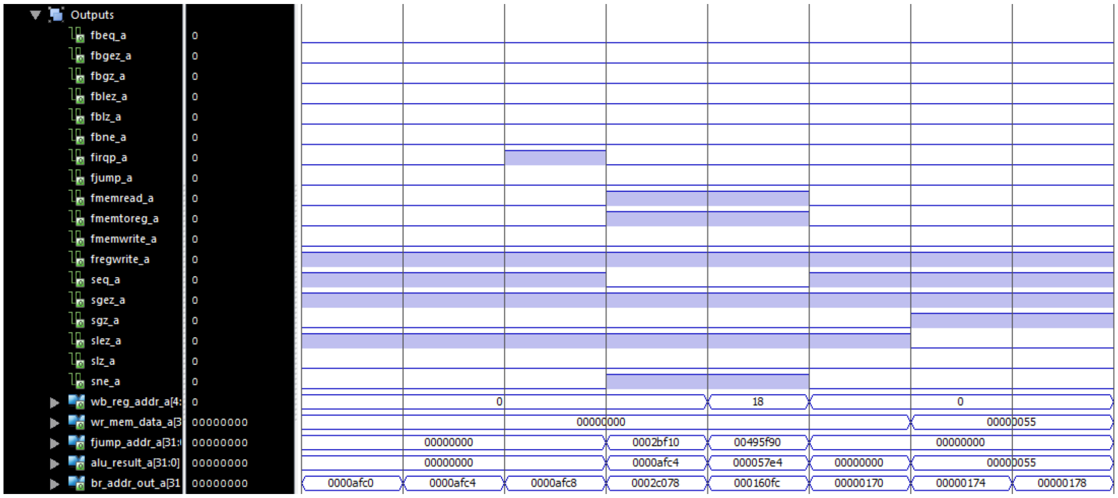



Outputs
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
wb_reg_addr_a[4:0]
wr_mem_data_a[31
fjump_addr_a[31:0]
alu_result_a[31:0]
br_addr_out_a[31:0

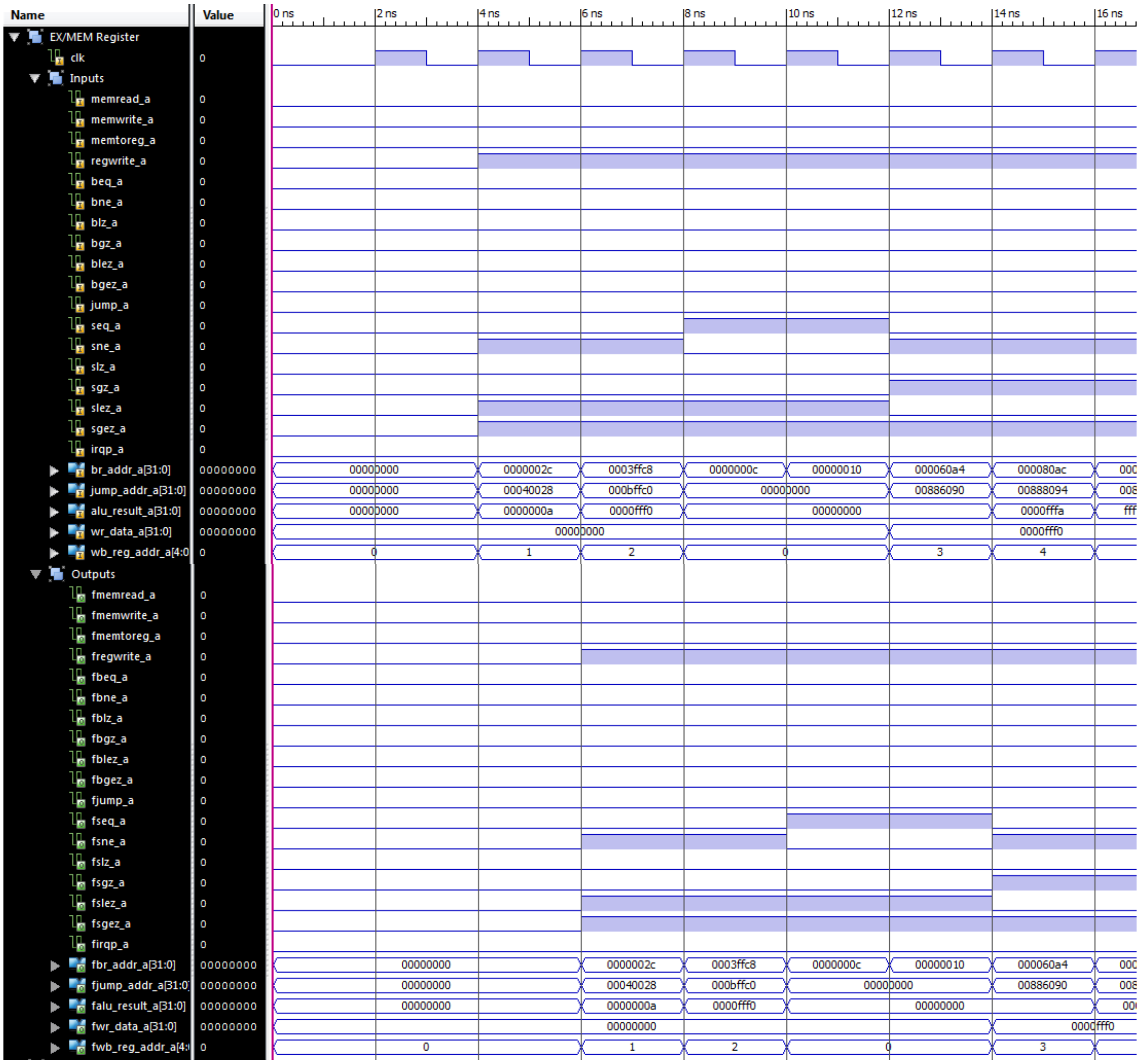

Name
Value
288 ns
290 ns
292 ns
294 ns
296 ns
298 ns
300 ns
302 ns
304 ns
EX Stage
Inputs
alusrc_a
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_a
memread_a
memtoreg_a
memwrite_a
regdst_a
regwrite_a
aluop_a[3:0]
shamt_a[4:0]
wb_reg_addr1_a[4:
wb_reg_addr2_a[4:
addr_in_a[31:0]
reg_op1_a[31:0]
reg_op2_a[31:0]
imm_op2_a[31:0]
jump_addr_a[31:0]
Outputs
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
wb_reg_addr_a[4:0]
wr_mem_data_a[31
fjump_addr_a[31:0]
alu_result_a[31:0]
br_addr_out_a[31:0

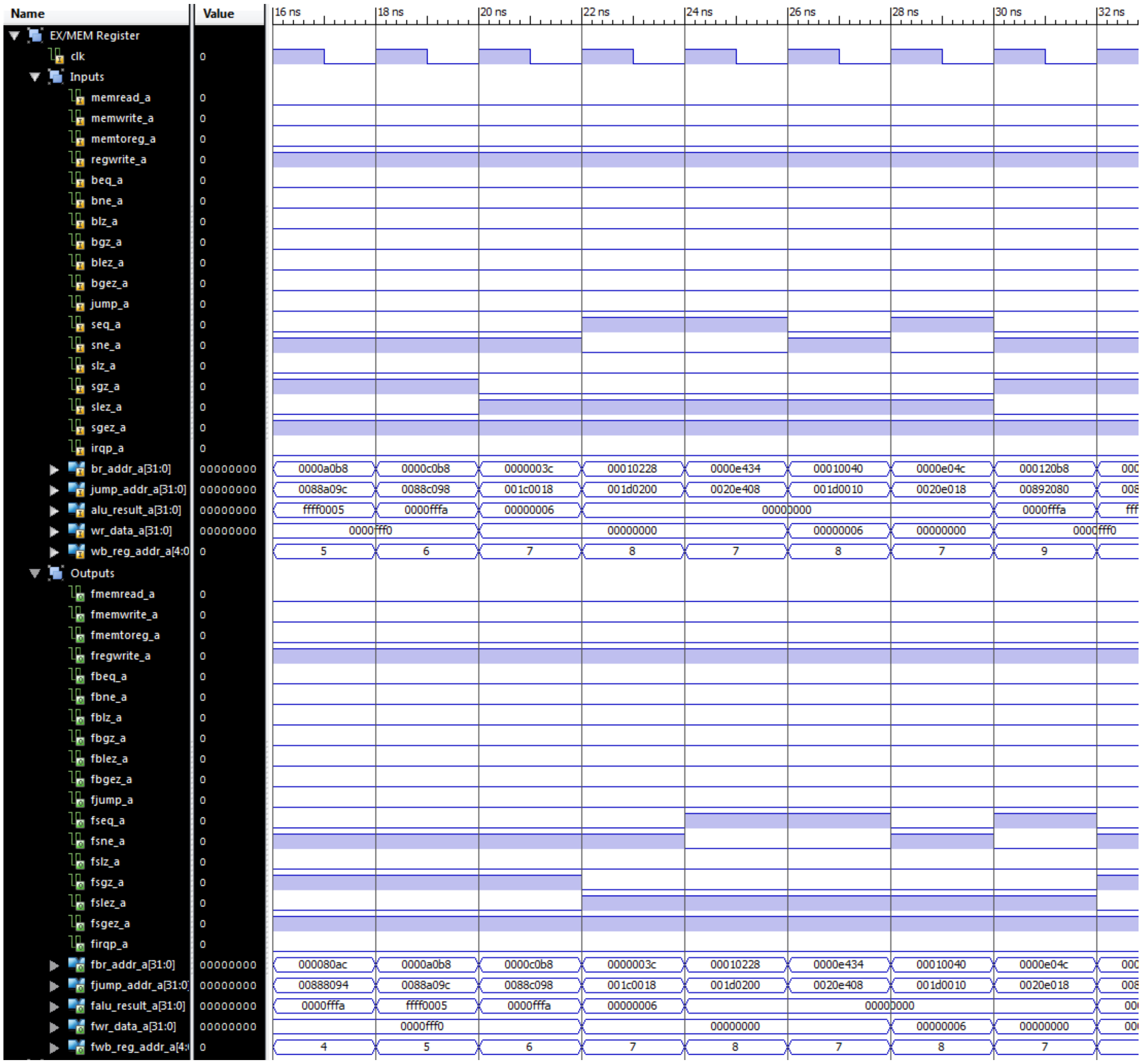


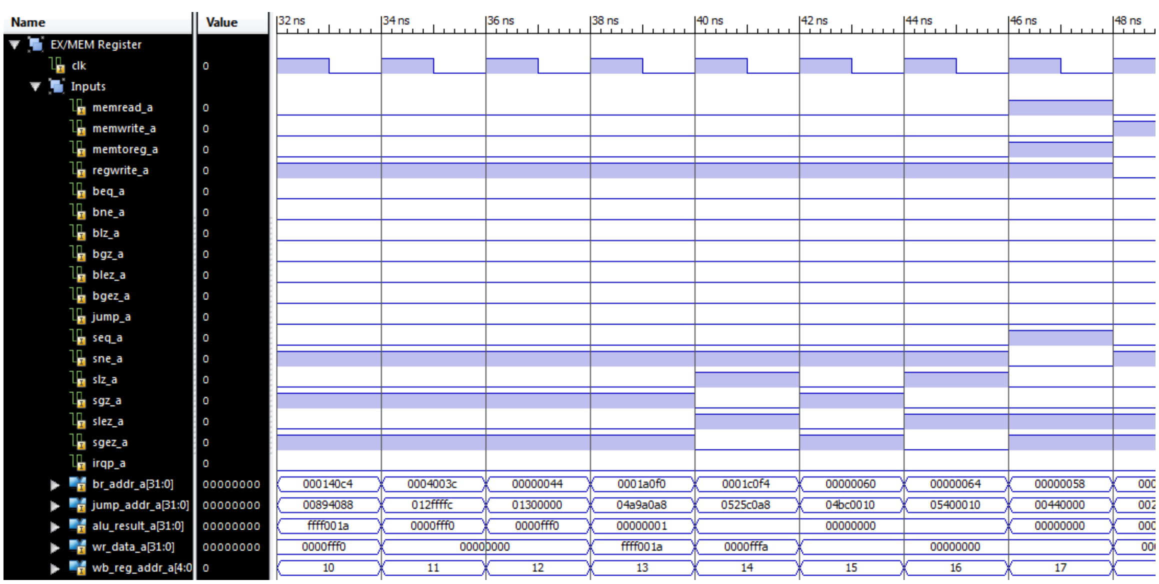


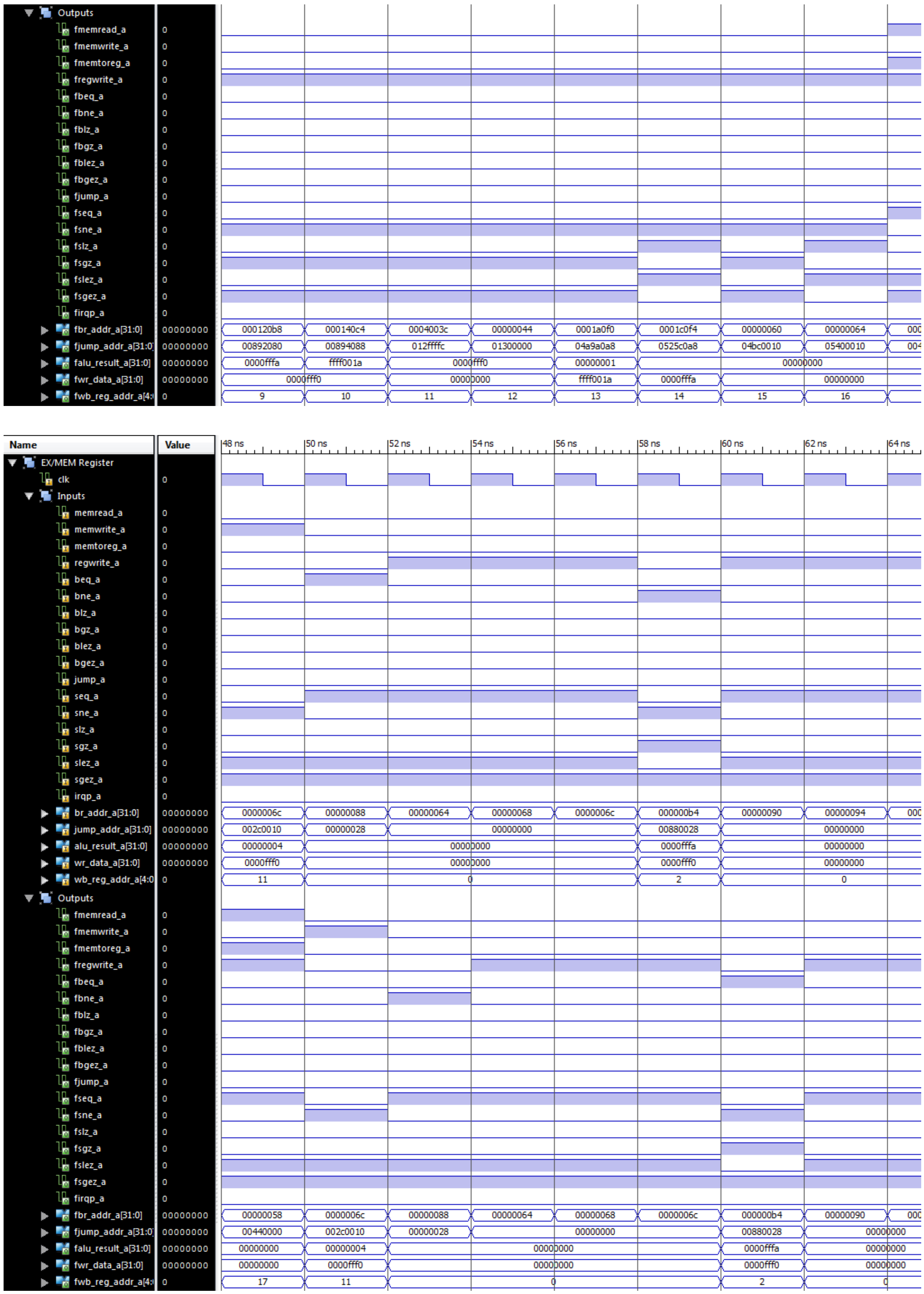

Outputs
fbeq_a
fbgez_a
fbgz_a
fblez_a
fblz_a
fbne_a
firqp_a
fjump_a
fmemread_a
fmemtoreg_a
fmemwrite_a
fregwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
wb_reg_addr_a[4:
wr_mem_data_a[3
fjump_addr_a[31:
alu_result_a[31:0]
br_addr_out_a[31
0
18
00000000
00000055
0002bf10
00495f90
0000afc4
000057e4
00000000
0000afc0
0000afc8
0002c078
000160fc
00000170
00000174
00000178

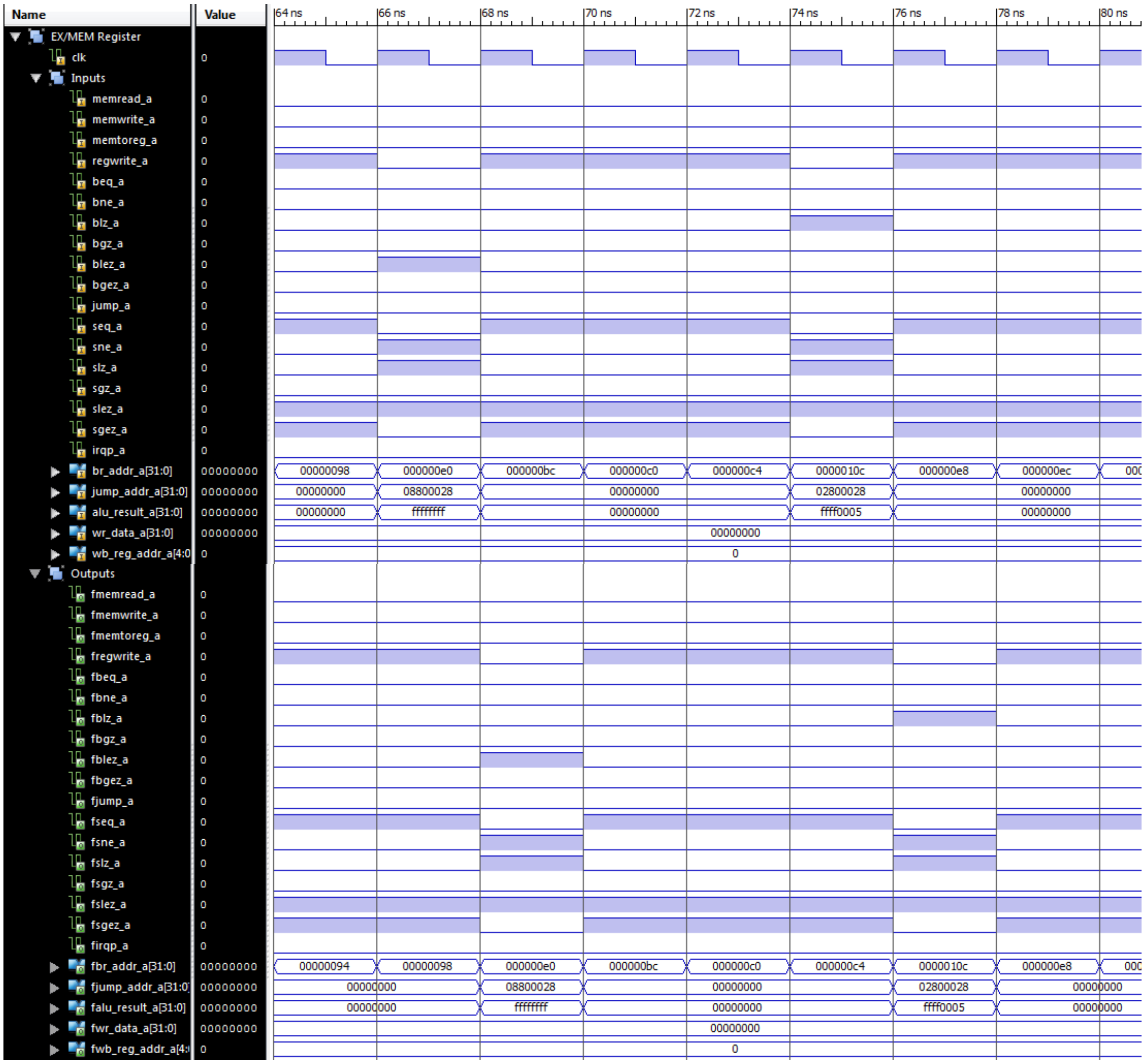
I. EX/MEM REGISTER

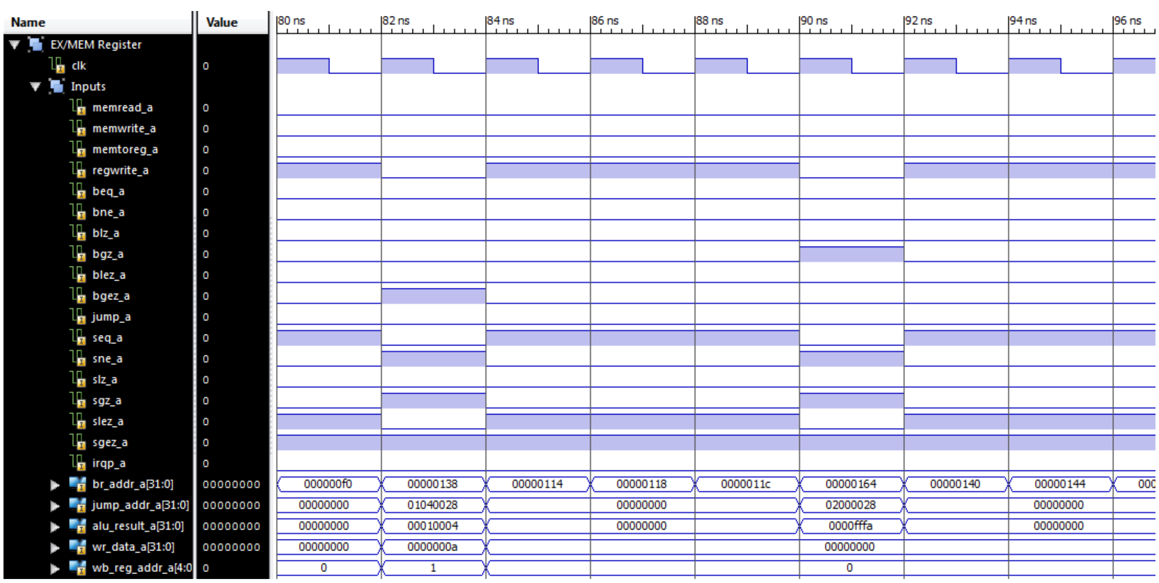


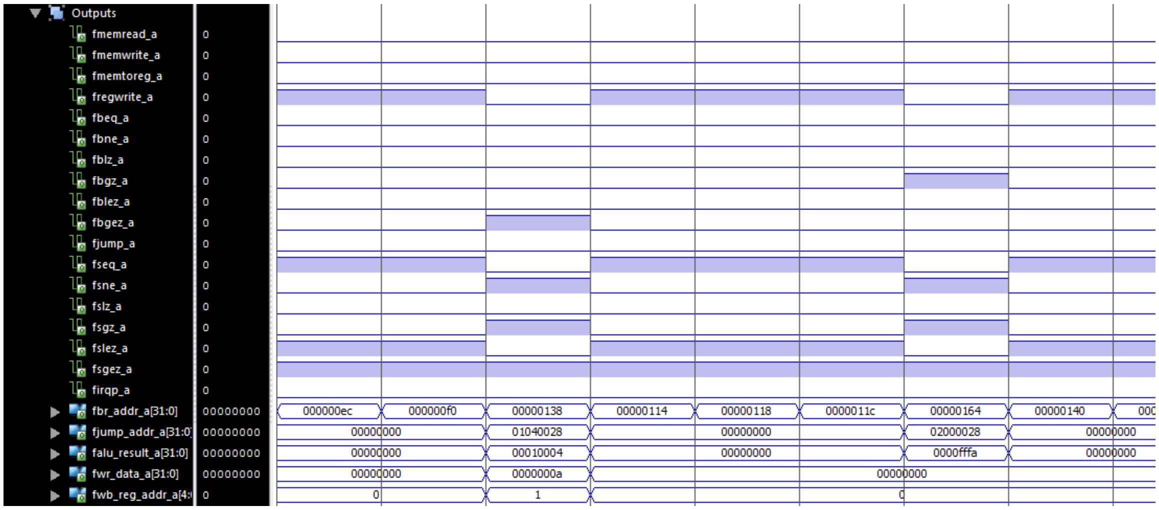


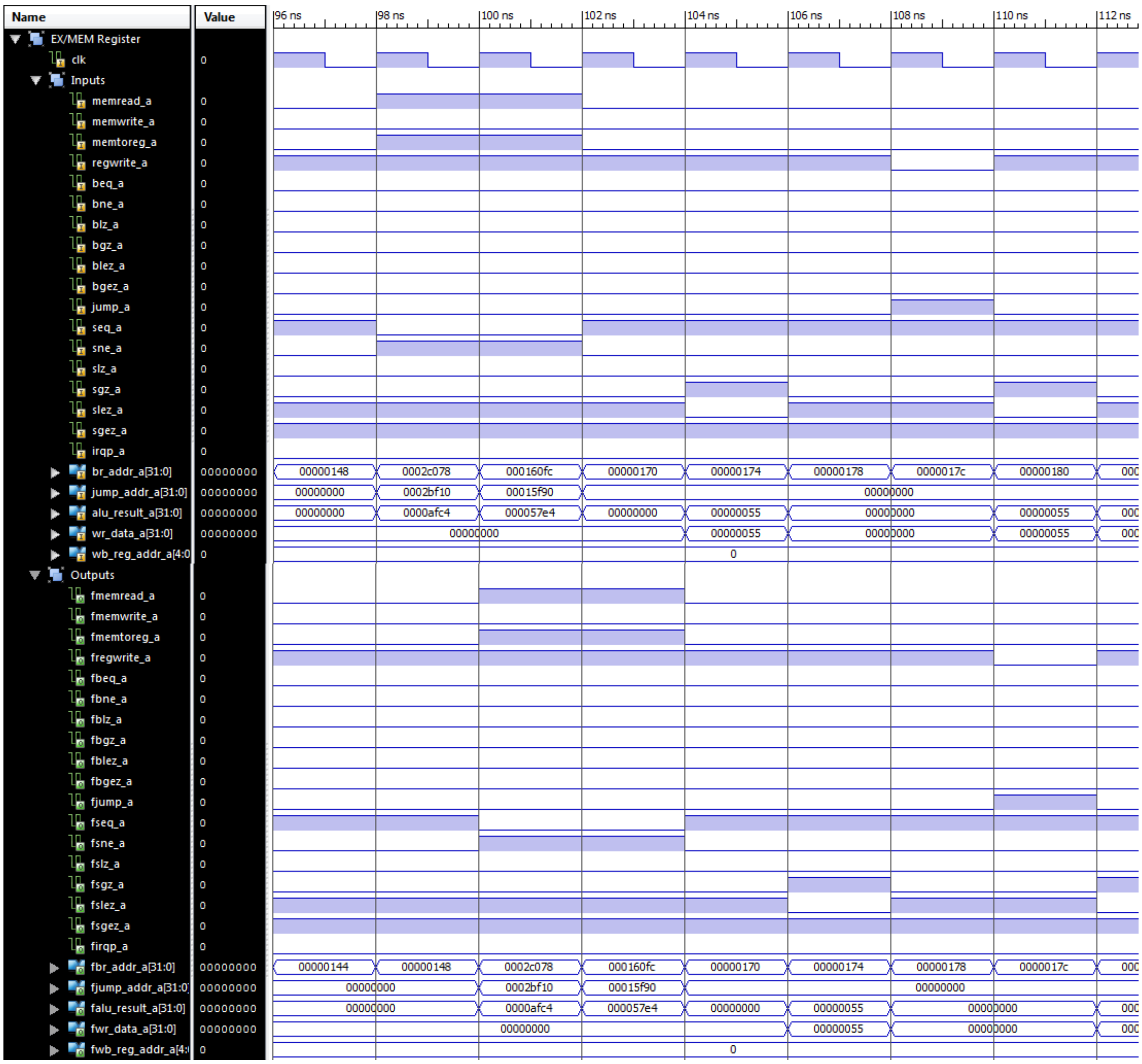


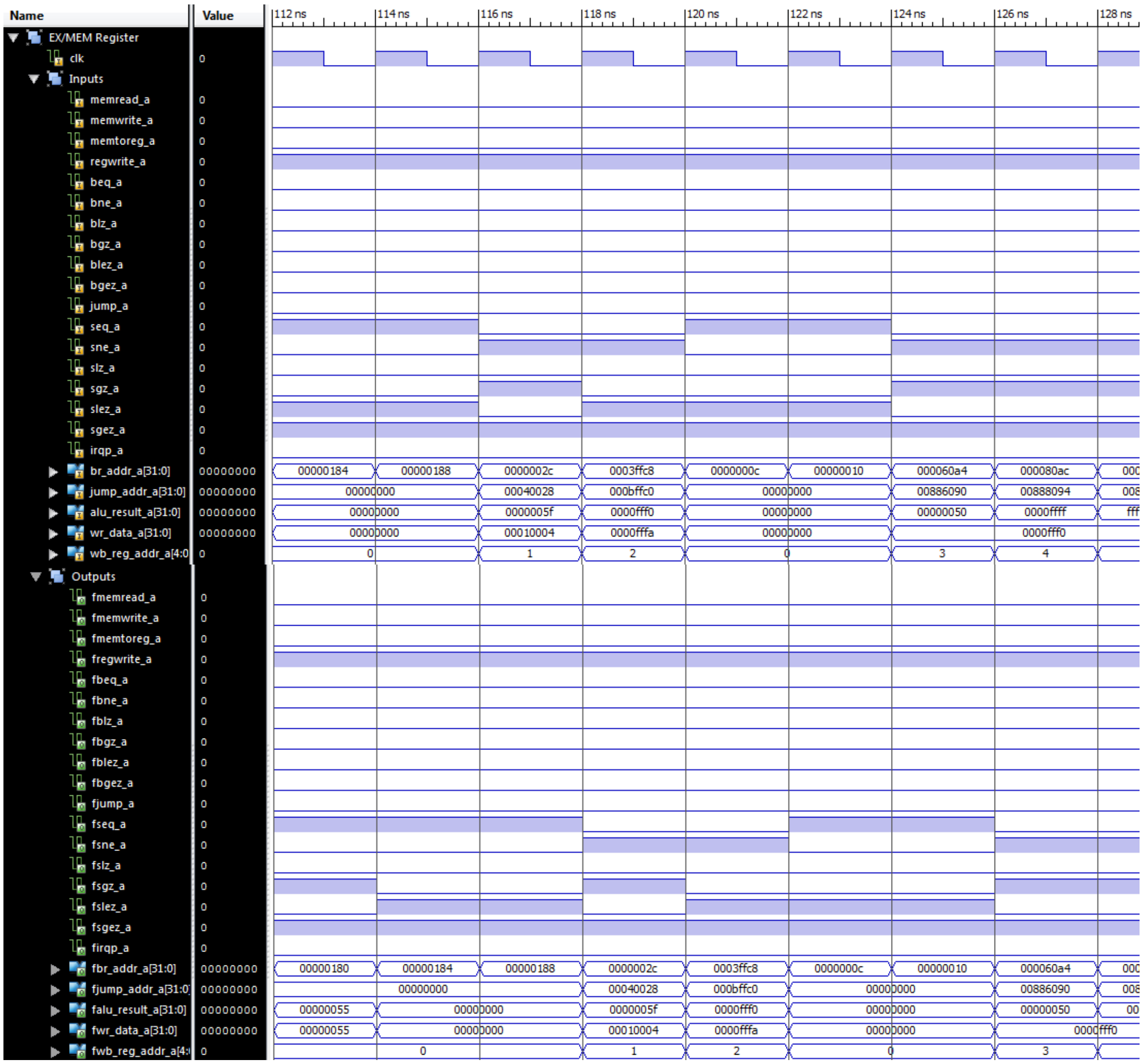

Outputs
fmemread_a
fmemwrite_a
fmemtoreg_a
fregwrite_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
fseq_a
fsne_a
fslz_a
fsgz_a
fslez_a
fsgez_a
firqp_a
fbr_addr_a[31:0]
fjump_addr_a[31:0]
falu_result_a[31:0]
fwr_data_a[31:0]
fwb_reg_addr_a[4:0]
000120b8 000140c4 0004003c 00000044 0001a0f0 0001c0f4 00000060 00000064
00892080 00894088 012ffffc 01300000 04a9a0a8 0525c0a8 04bc0010 05400010
0000fffa ffff001a 0000fff0 00000001 00000000
0000fff0 00000000 ffff001a 0000fffa 00000000
9 10 11 12 13 14 15 16
Name
Value
48 ns 50 ns 52 ns 54 ns 56 ns 58 ns 60 ns 62 ns 64 ns
EX/MEM Register
clk
Inputs
memread_a
memwrite_a
memtoreg_a
regwrite_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
seq_a
sne_a
slz_a
sgz_a
slez_a
sgez_a
irqp_a
br_addr_a[31:0]
jump_addr_a[31:0]
alu_result_a[31:0]
wr_data_a[31:0]
wb_reg_addr_a[4:0]
0000006c 00000088 00000064 00000068 0000006c 000000b4 00000090 00000094
002c0010 00000028 00000000 00880028 00000000
00000004 00000000 0000fffa 00000000
0000fff0 00000000 0000fff0 00000000
11 0 2 0
Outputs
fmemread_a
fmemwrite_a
fmemtoreg_a
fregwrite_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
fseq_a
fsne_a
fslz_a
fsgz_a
fslez_a
fsgez_a
firqp_a
fbr_addr_a[31:0]
fjump_addr_a[31:0]
falu_result_a[31:0]
fwr_data_a[31:0]
fwb_reg_addr_a[4:0]
00000058 0000006c 00000088 00000064 00000068 0000006c 000000b4 00000090
00440000 002c0010 00000028 00000000 00880028 00000000
00000000 00000004 00000000 0000fffa 00000000
00000000 0000fff0 00000000 0000fff0 00000000
17 11 0 2 0

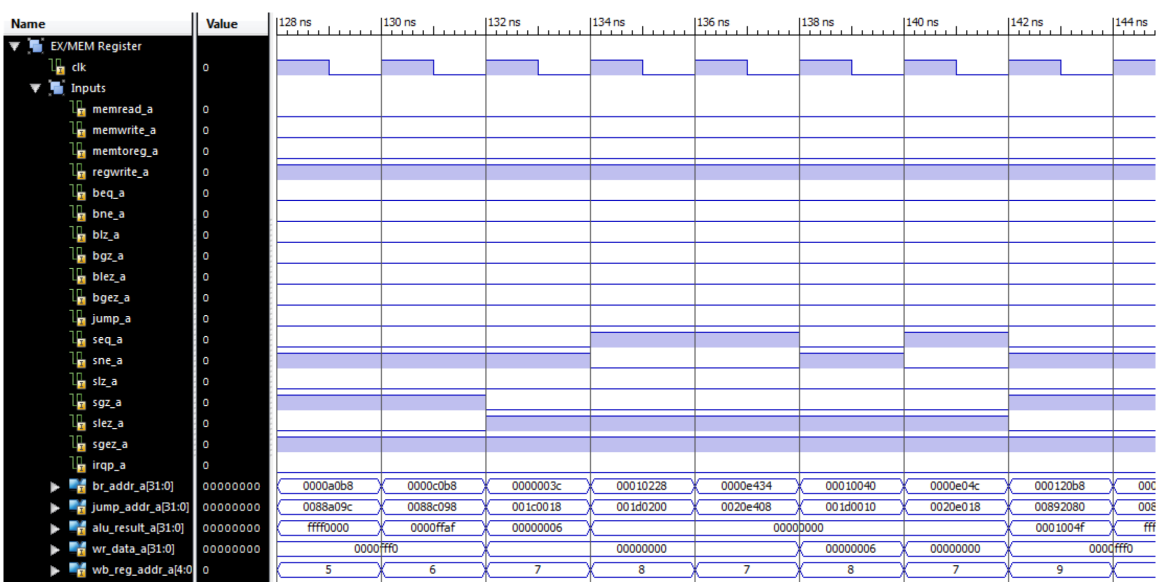

Name
Value
EX/MEM Register
clk
Inputs
memread_a
memwrite_a
memtoreg_a
regwrite_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
seq_a
sne_a
slz_a
sgz_a
slez_a
sgez_a
irqp_a
br_addr_a[31:0]
jump_addr_a[31:0]
alu_result_a[31:0]
wr_data_a[31:0]
wb_reg_addr_a[4:0]
Outputs
fmemread_a
fmemwrite_a
fmemtoreg_a
fregwrite_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
fseq_a
fsne_a
fslz_a
fsgz_a
fslez_a
fsgez_a
firqp_a
fbr_addr_a[31:0]
fjump_addr_a[31:0]
falu_result_a[31:0]
fwr_data_a[31:0]
fwb_reg_addr_a[4:0]
64 ns
66 ns
68 ns
70 ns
72 ns
74 ns
76 ns
78 ns
80 ns
00000098
000000e0
000000bc
000000c0
000000c4
0000010c
000000e8
000000ec
00000000
08800028
02800028
ffffffff
ffff0005
0
00000094

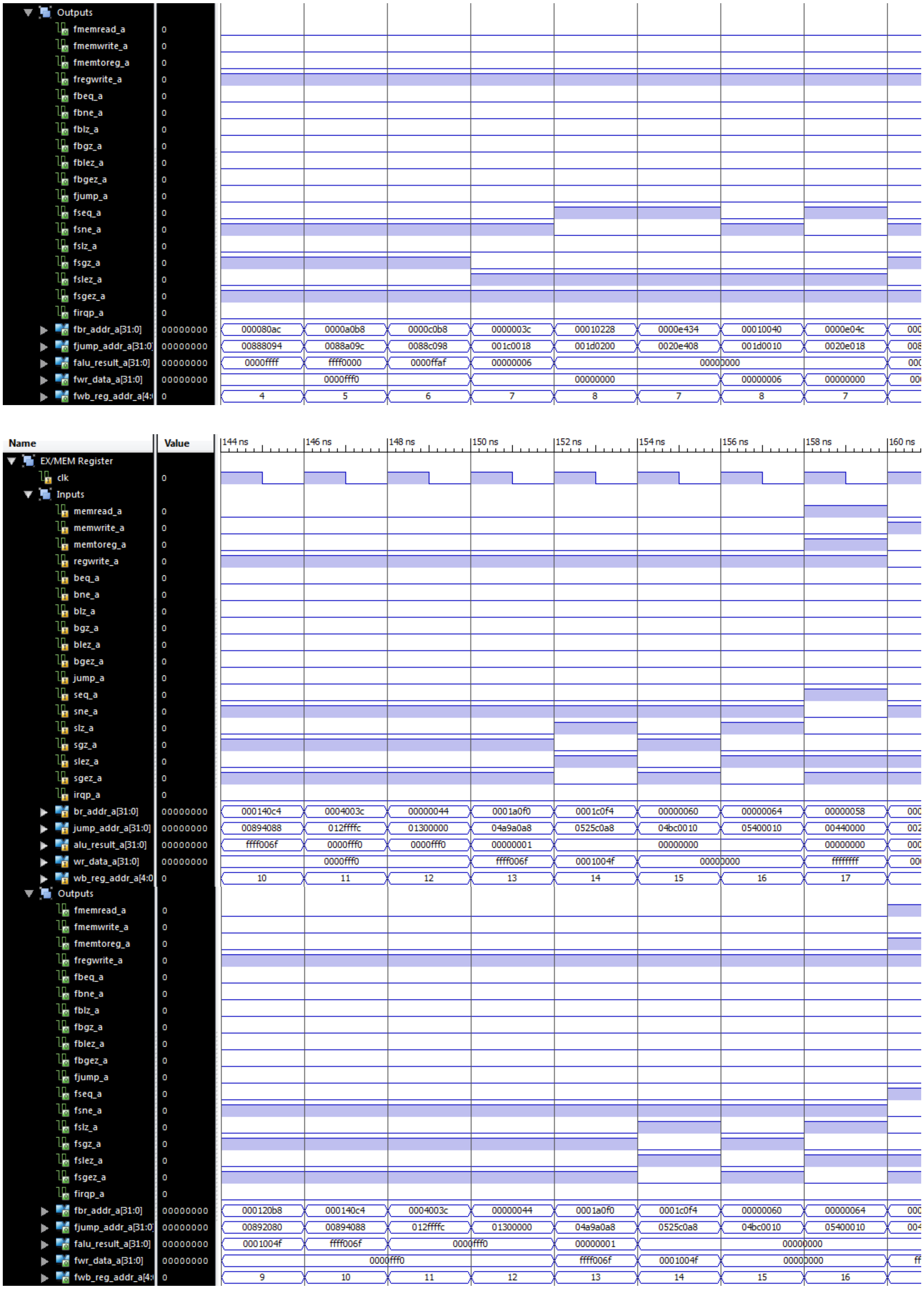

Name
Value
EX/MEM Register
clk
Inputs
memread_a
memwrite_a
memtoreg_a
regwrite_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
seq_a
sne_a
slz_a
sgz_a
slez_a
sgez_a
irqp_a
br_addr_a[31:0]
jump_addr_a[31:0]
alu_result_a[31:0]
wr_data_a[31:0]
wb_reg_addr_a[4:0]
80 ns
82 ns
84 ns
86 ns
88 ns
90 ns
92 ns
94 ns
96 ns
000000f0
00000138
00000114
00000118
0000011c
00000164
00000140
00000144
00000000
01040028
02000028
00010004
0000fffa
0000000a
0
1

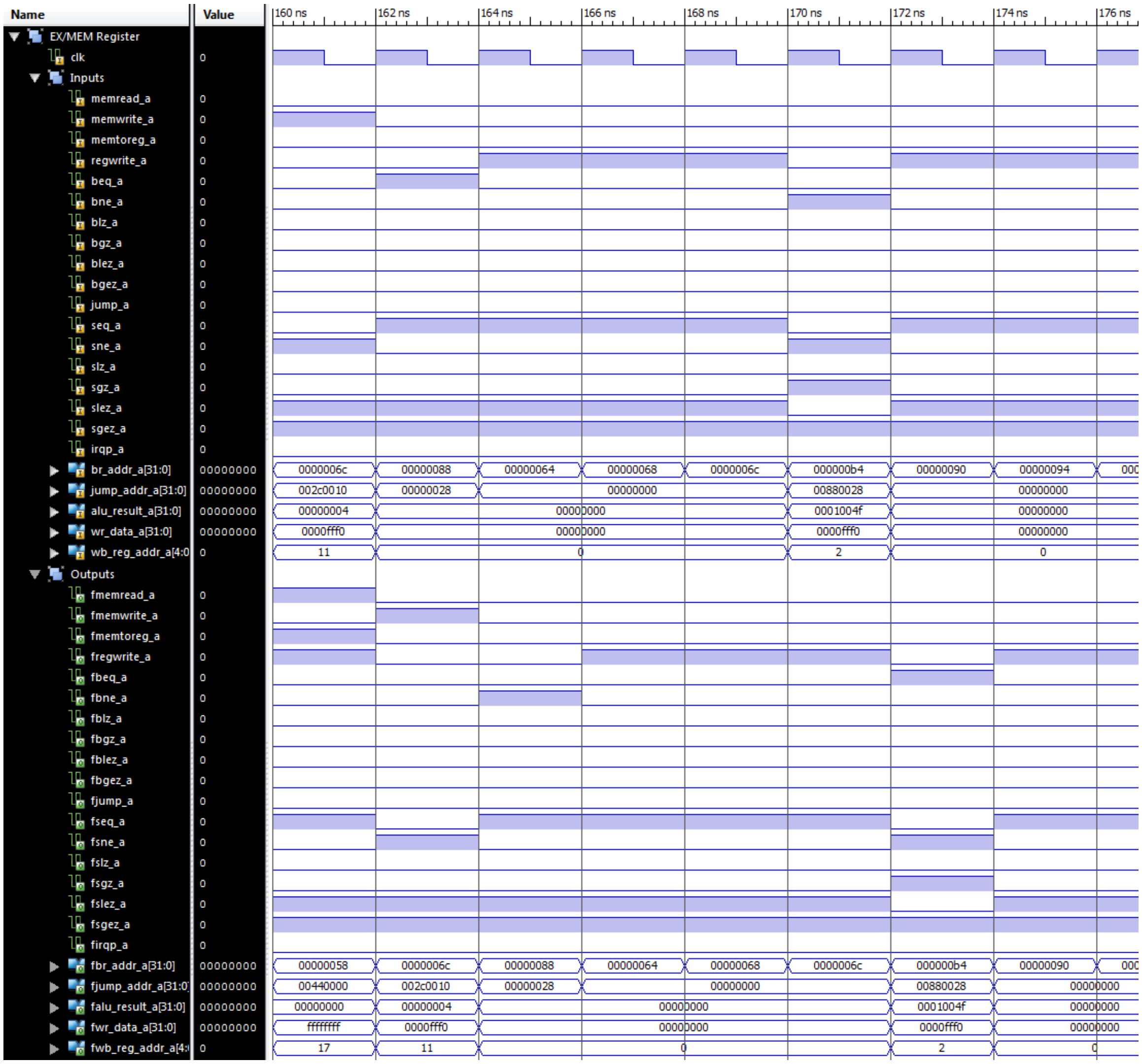


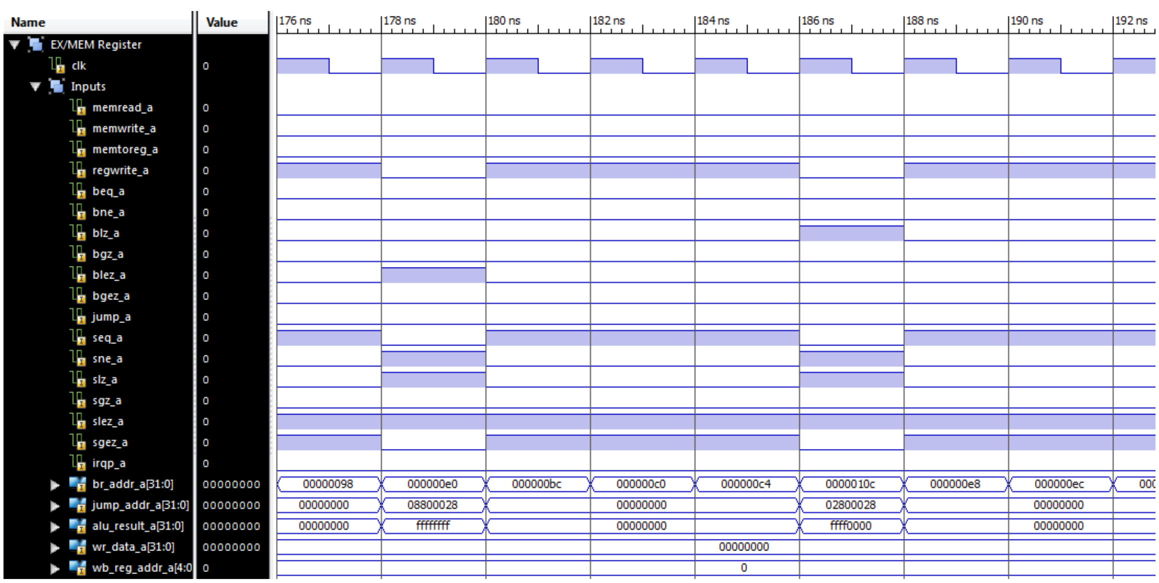


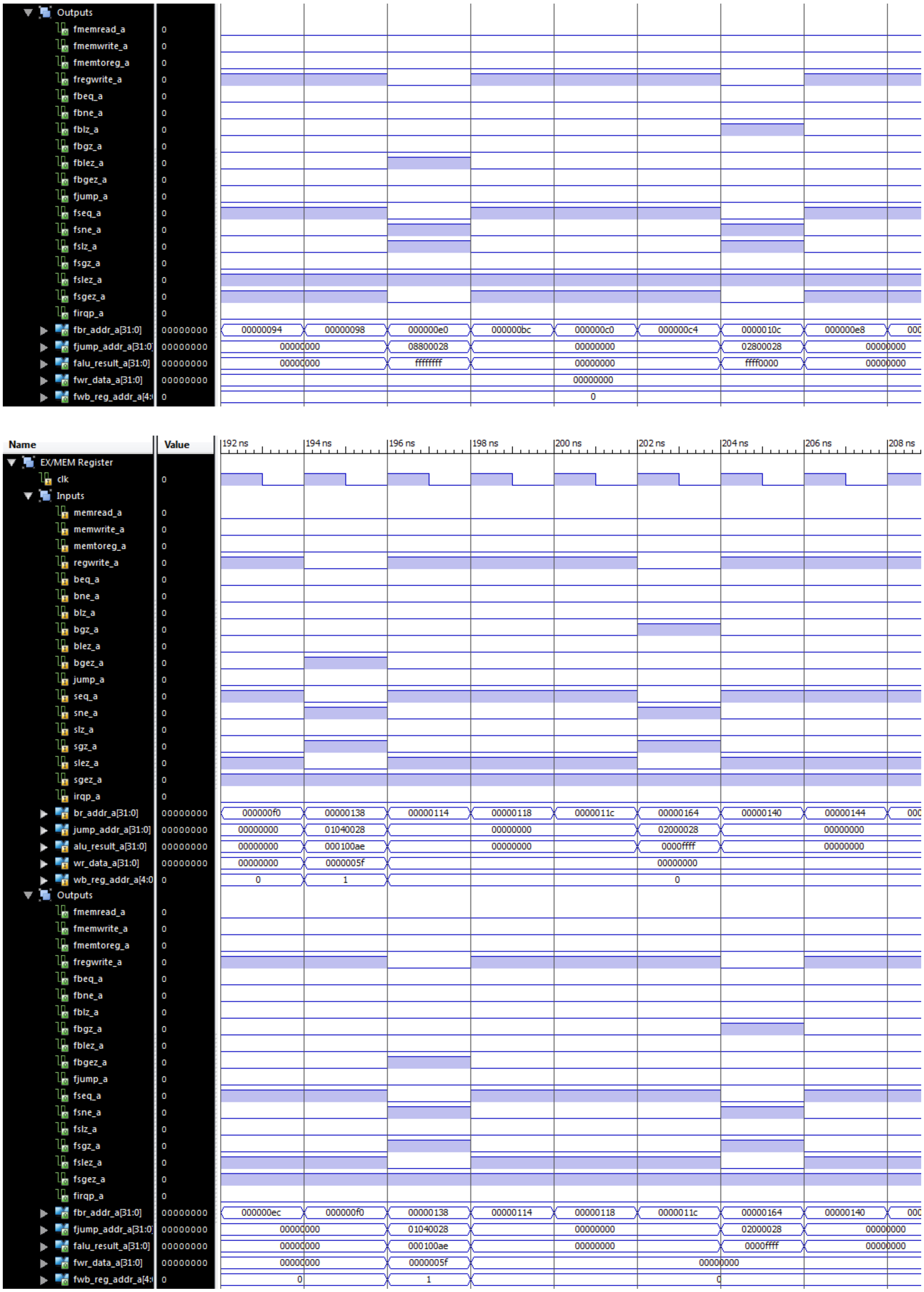


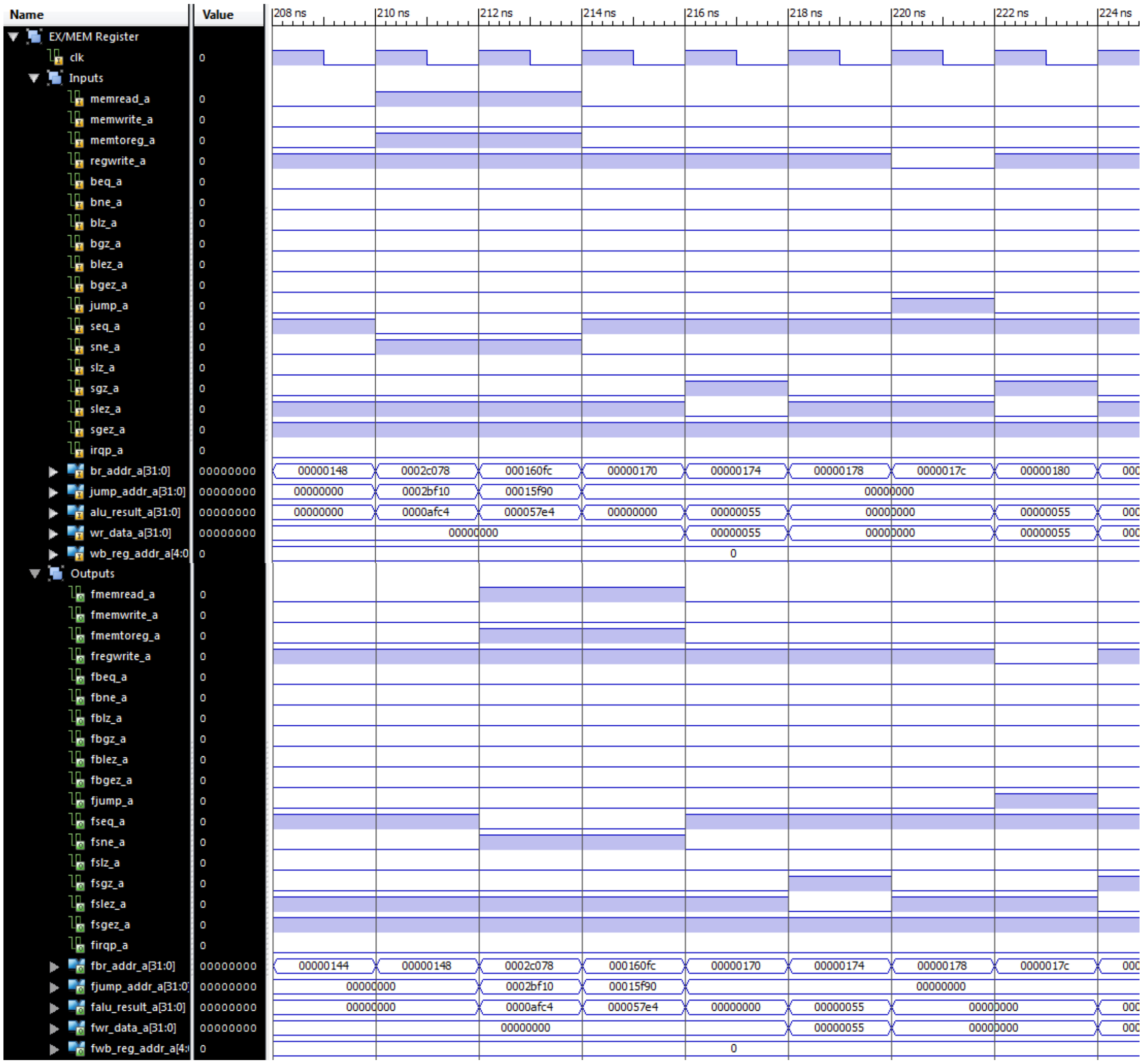


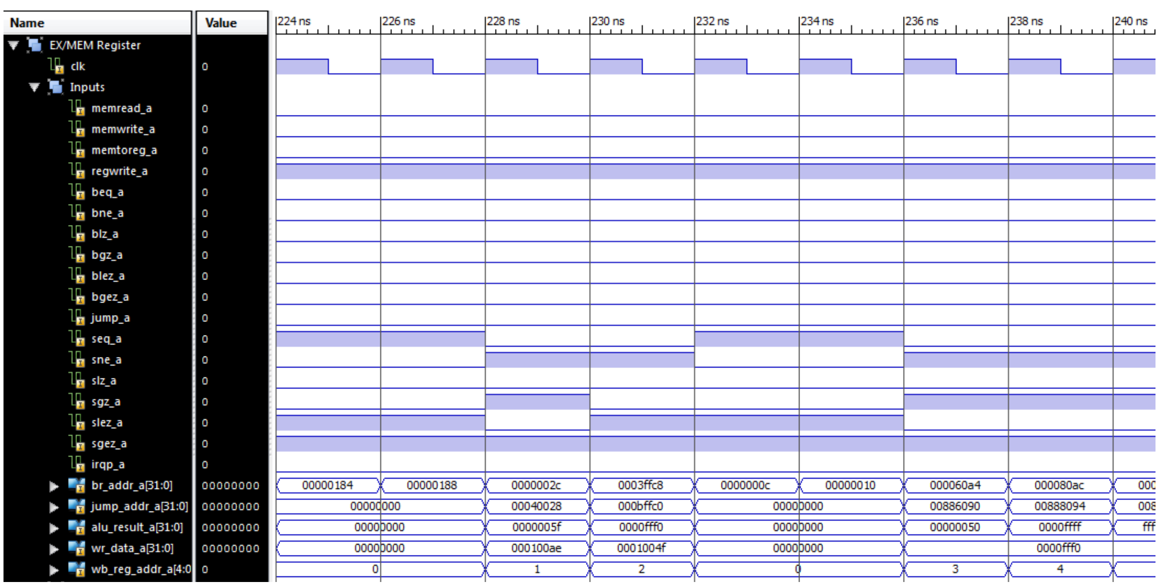


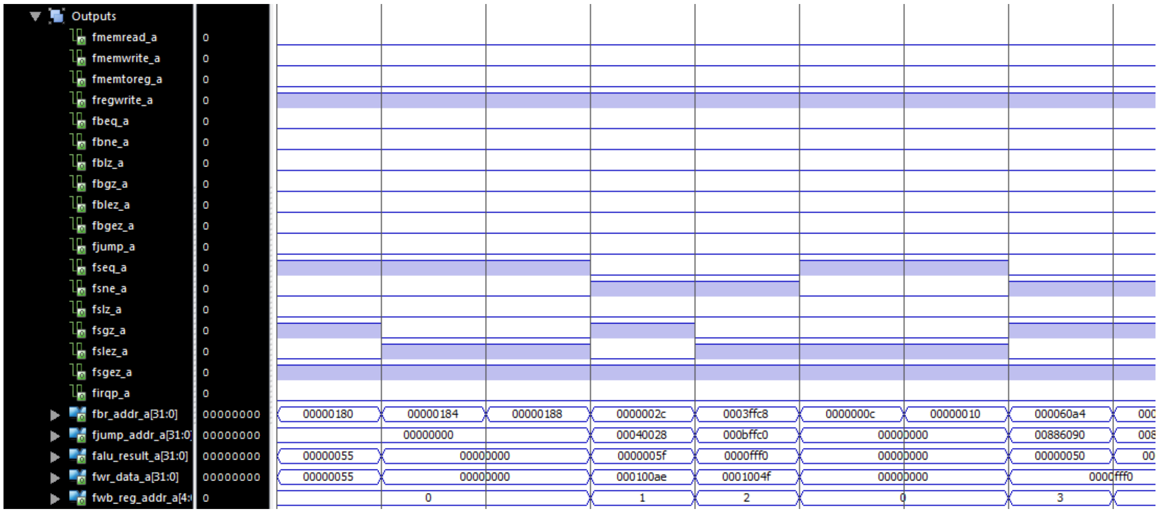


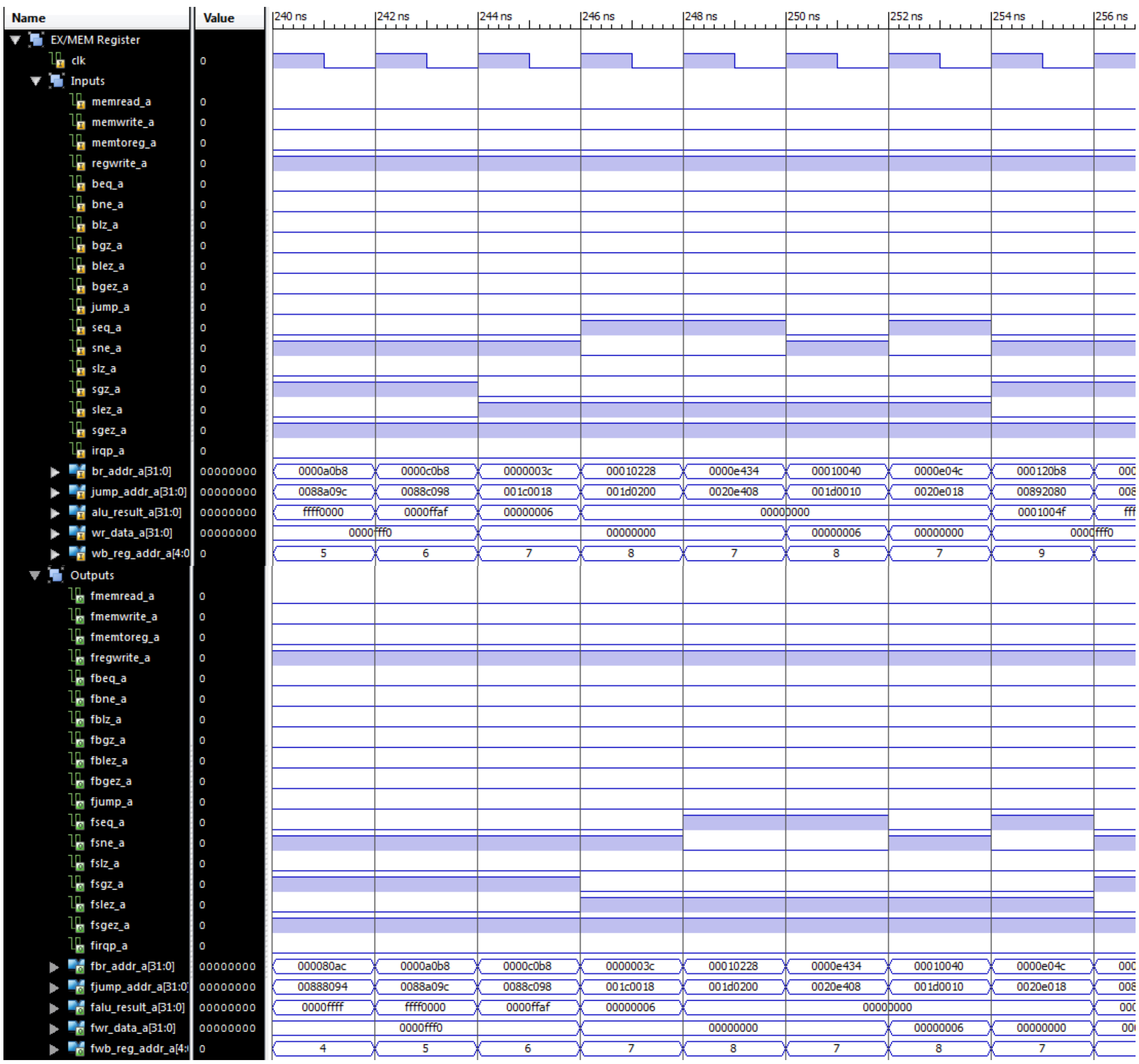


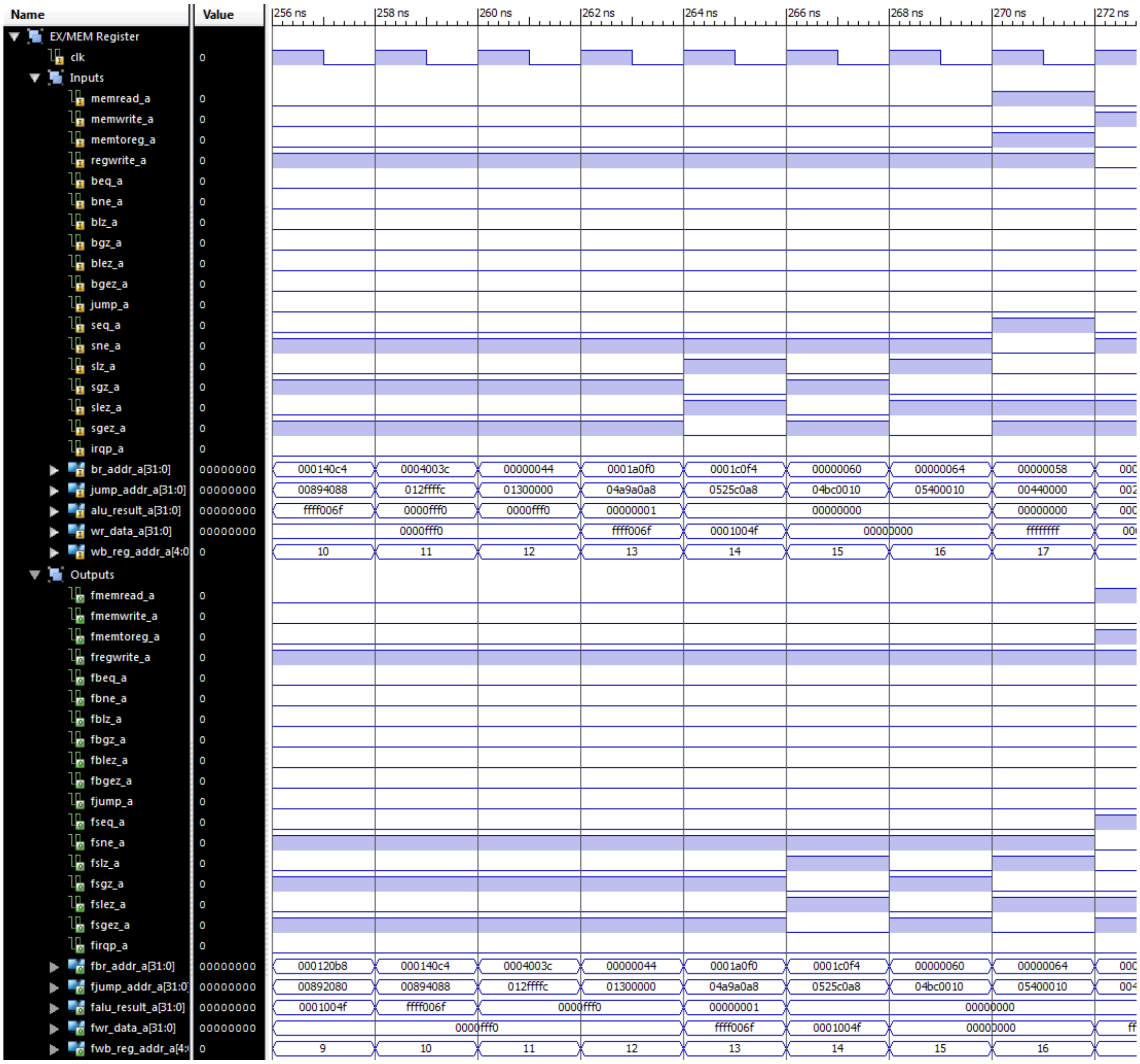


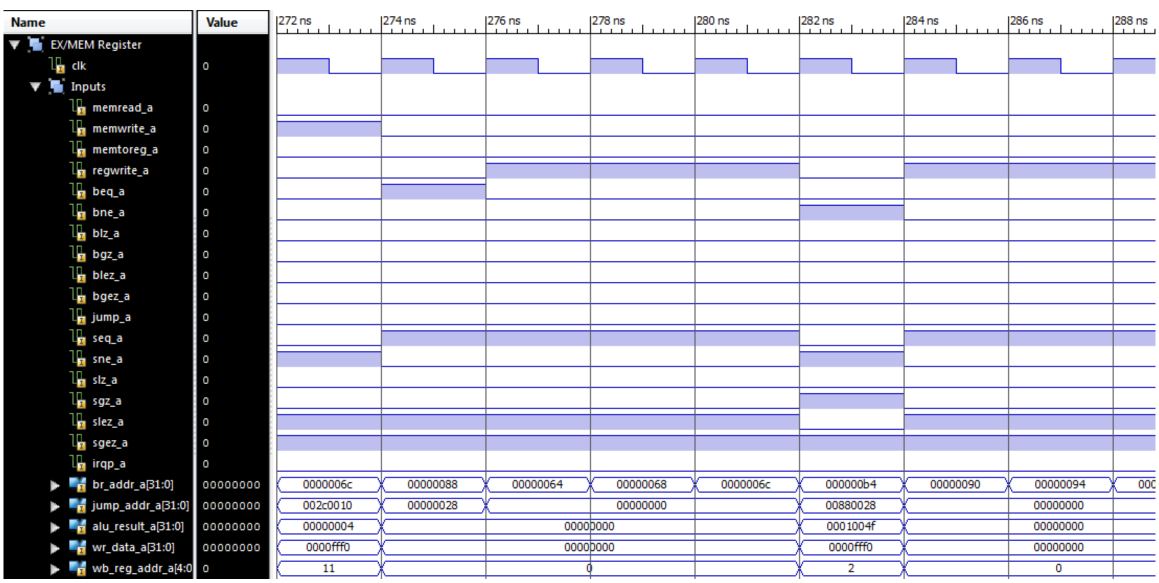


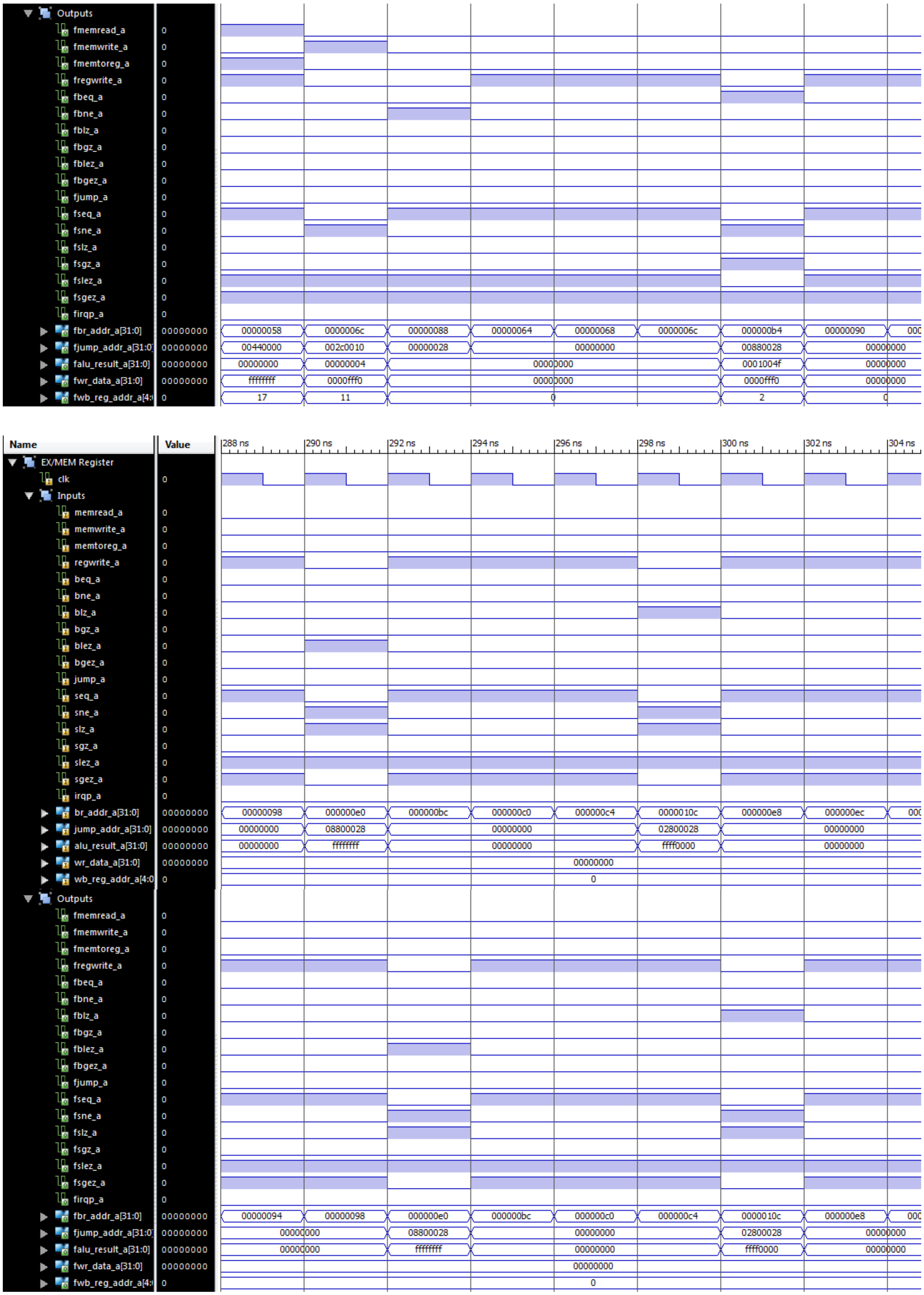


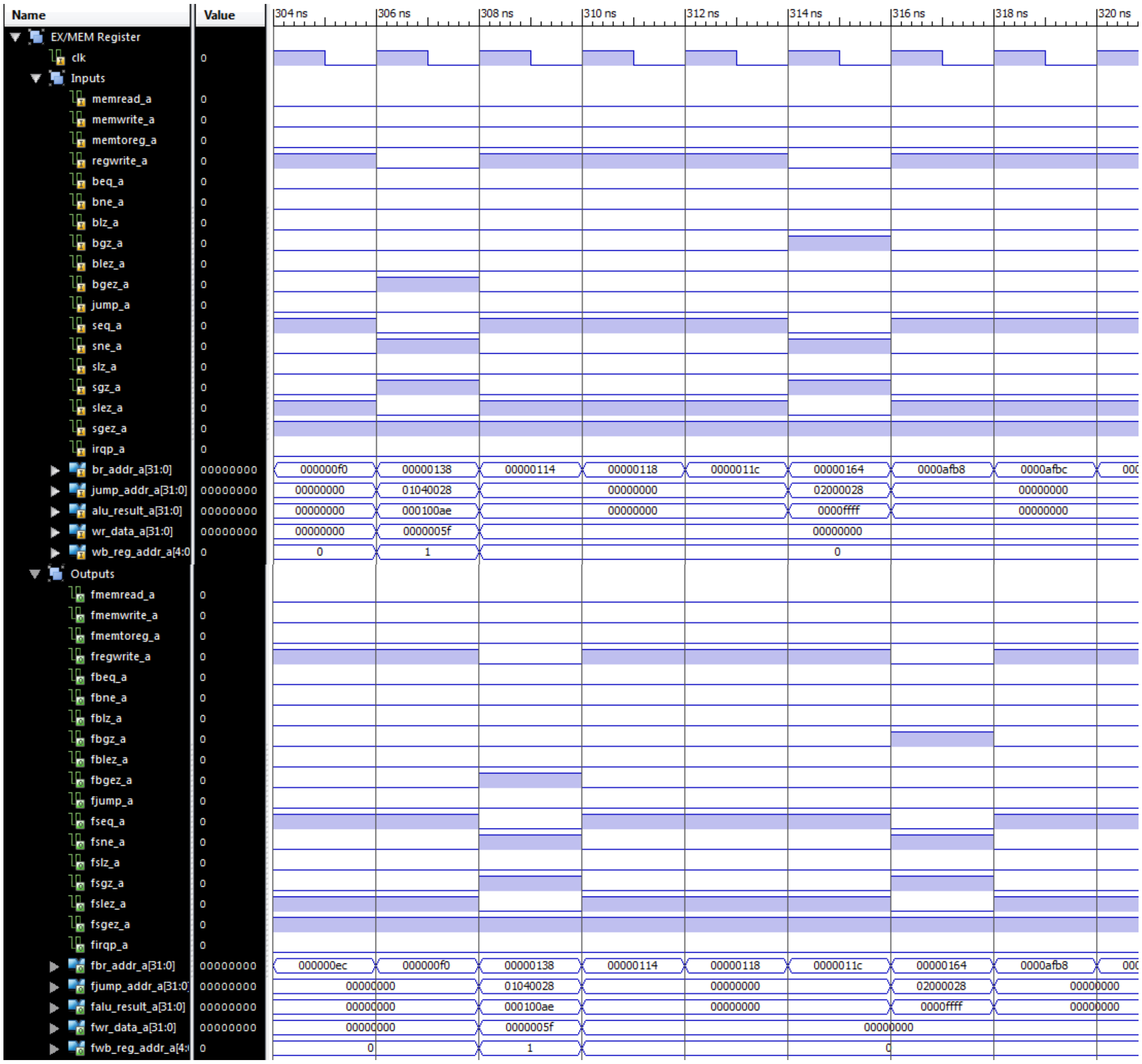


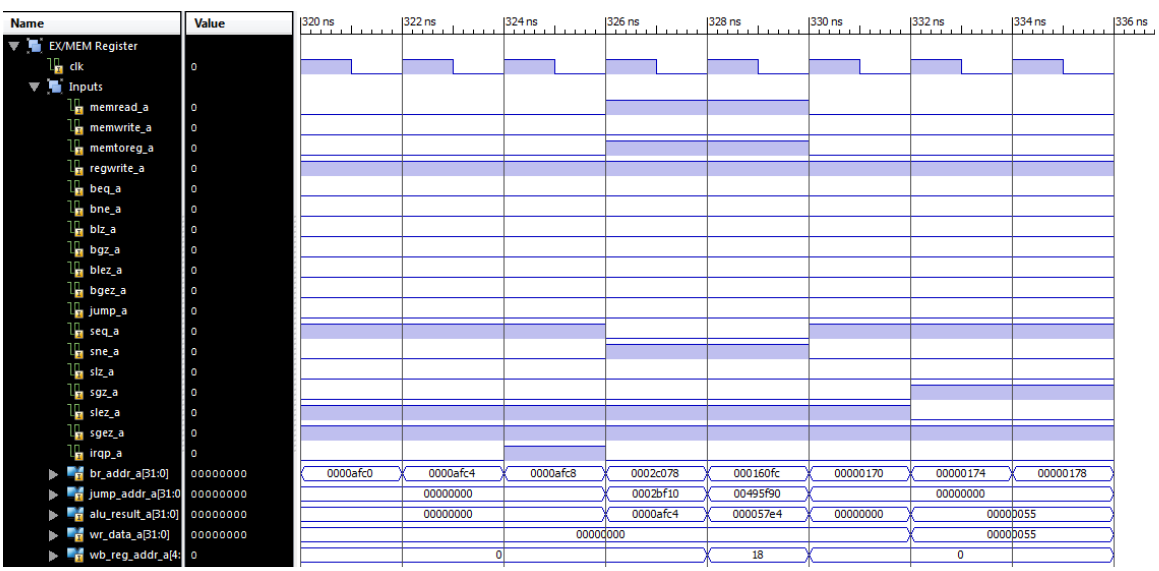


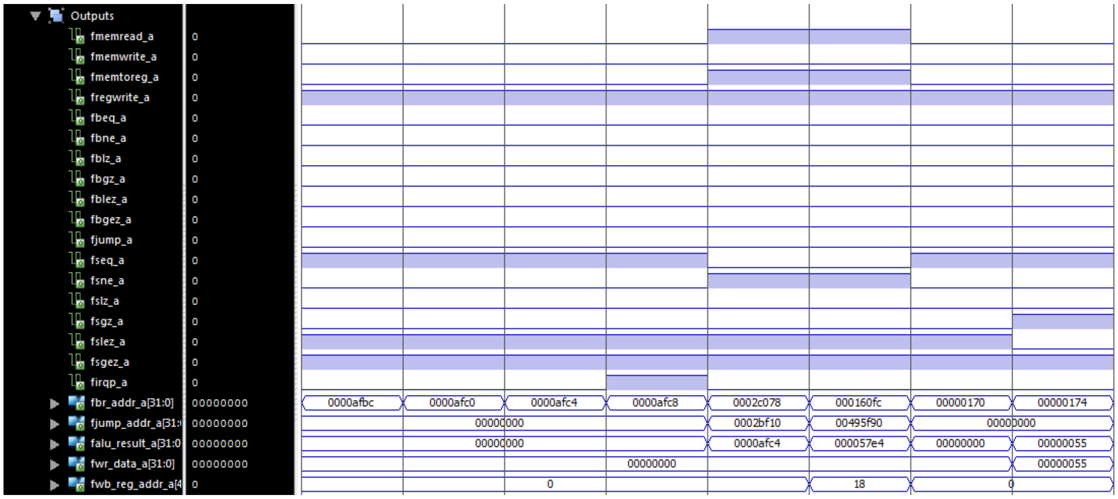




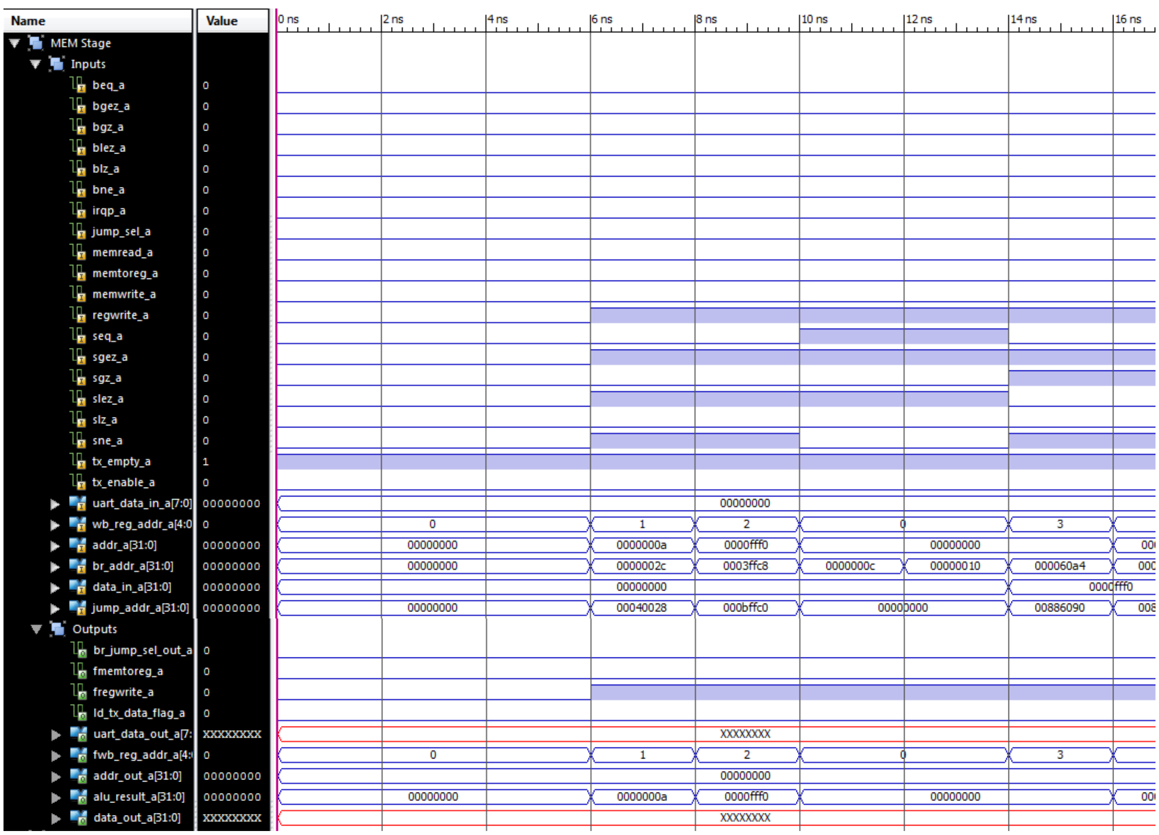


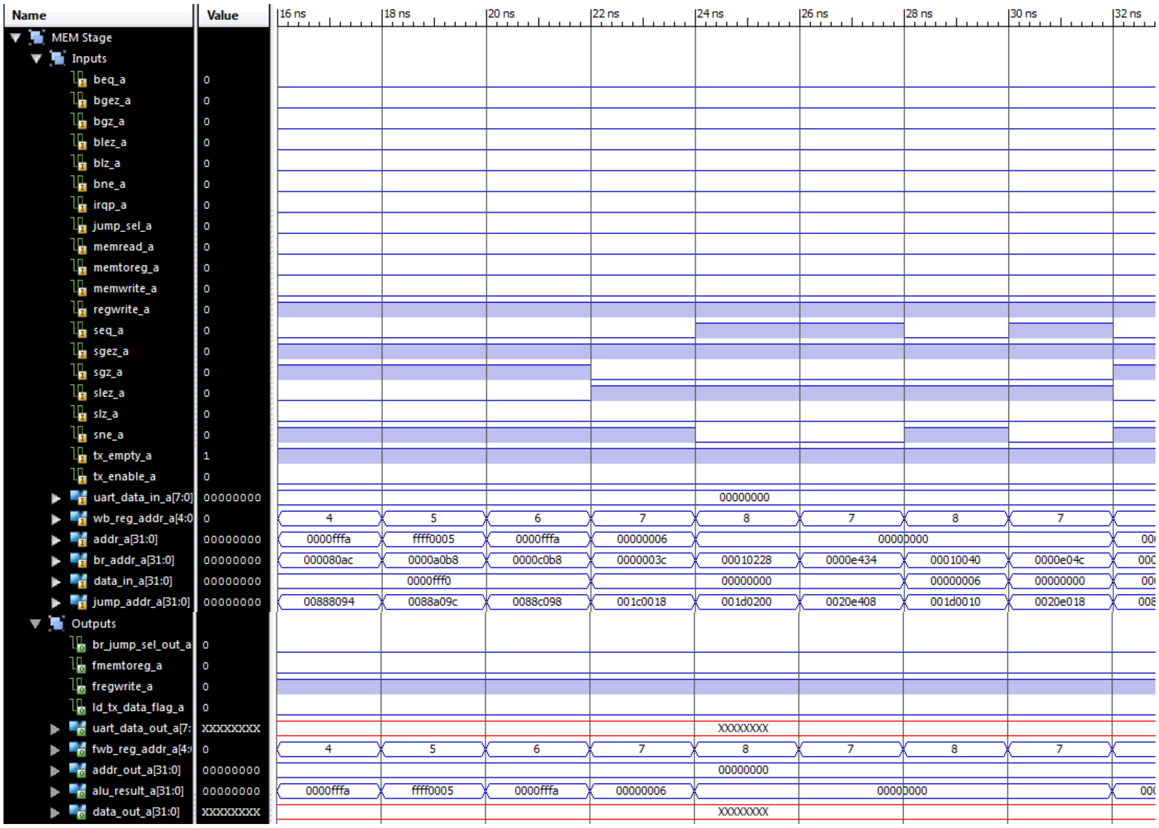

Name
Value
304 ns
306 ns
308 ns
310 ns
312 ns
314 ns
316 ns
318 ns
320 ns
EX/MEM Register
clk
Inputs
memread_a
memwrite_a
memtoreg_a
regwrite_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
seq_a
sne_a
slz_a
sgz_a
slez_a
sgez_a
irqp_a
br_addr_a[31:0]
jump_addr_a[31:0]
alu_result_a[31:0]
wr_data_a[31:0]
wb_reg_addr_a[4:0]
000000f0
00000138
00000114
00000118
0000011c
00000164
0000afb8
0000afbc
01040028
02000028
000100ae
0000ffff
0000005f
00000000
Outputs
fmemread_a
fmemwrite_a
fmemtoreg_a
fregwrite_a
fbeq_a
fbne_a
fblz_a
fbgz_a
fblez_a
fbgez_a
fjump_a
fseq_a
fsne_a
fslz_a
fsgz_a
fslez_a
fsgez_a
firqp_a
fbr_addr_a[31:0]
fjump_addr_a[31:0]
falu_result_a[31:0]
fwr_data_a[31:0]
fwb_reg_addr_a[4:0]
000000ec
000000f0
00000138
00000114
00000118
0000011c
00000164
0000afb8
01040028
02000028
000100ae
0000ffff
0000005f
00000000

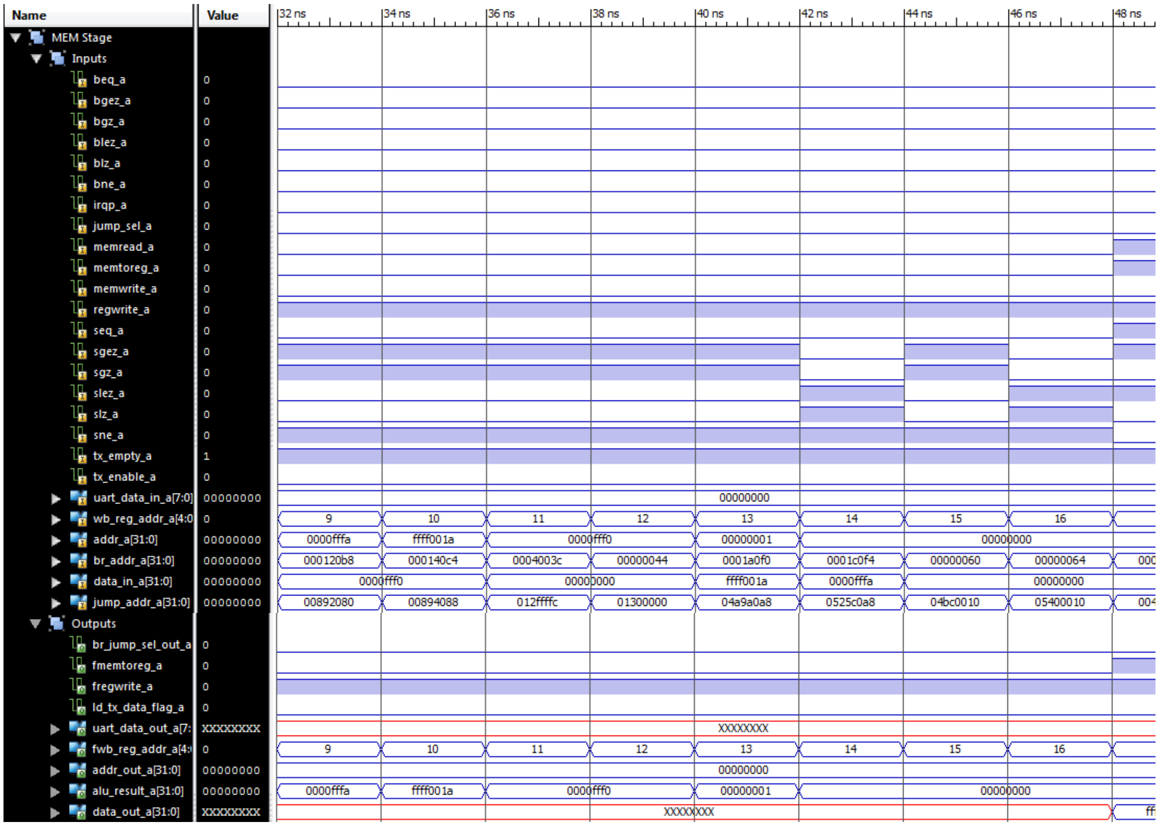

Name
Value
320 ns
322 ns
324 ns
326 ns
328 ns
330 ns
332 ns
334 ns
336 ns
EX/MEM Register
clk
Inputs
memread_a
memwrite_a
memtoreg_a
regwrite_a
beq_a
bne_a
blz_a
bgz_a
blez_a
bgez_a
jump_a
seq_a
sne_a
slz_a
sgz_a
slez_a
sgez_a
irqp_a
br_addr_a[31:0]
jump_addr_a[31:0]
alu_result_a[31:0]
wr_data_a[31:0]
wb_reg_addr_a[4:0]
0000afc0
0000afc4
0000afc8
0002c078
000160fc
00000170
00000174
00000178
0002bf10
00495f90
000057e4
00000055
00000000
18

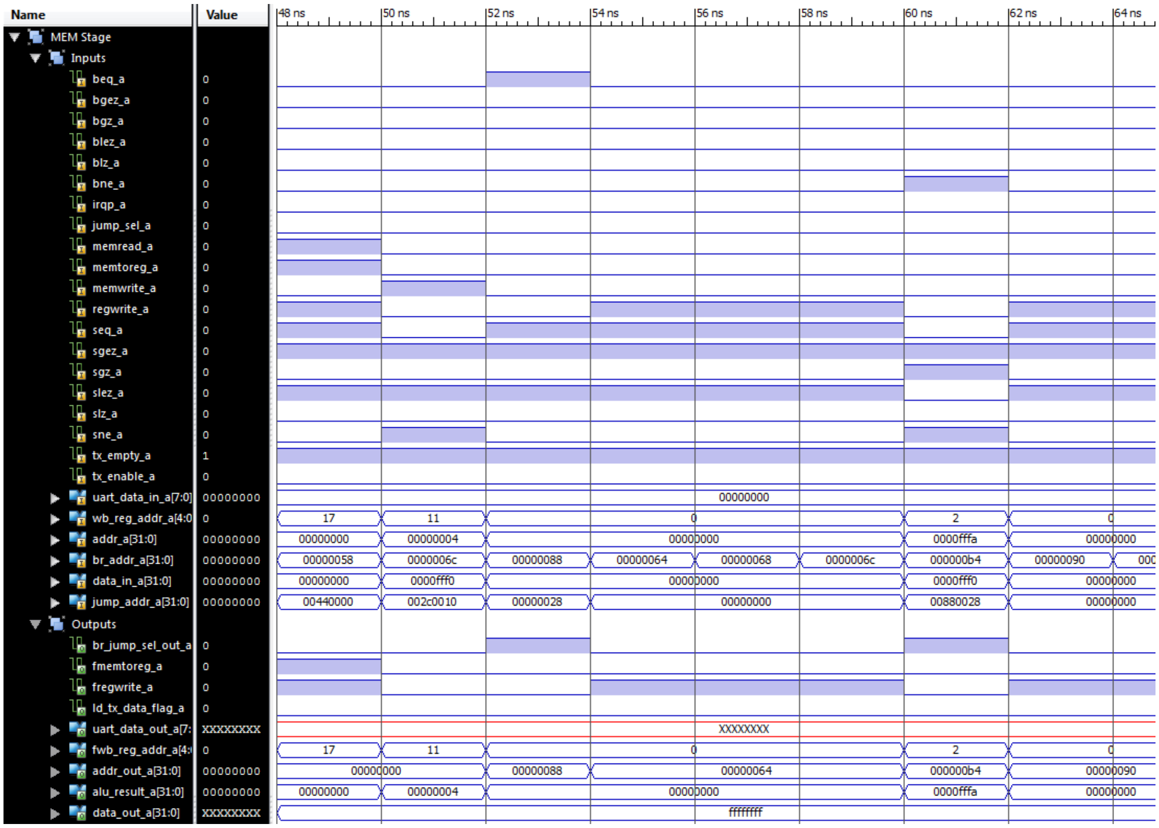


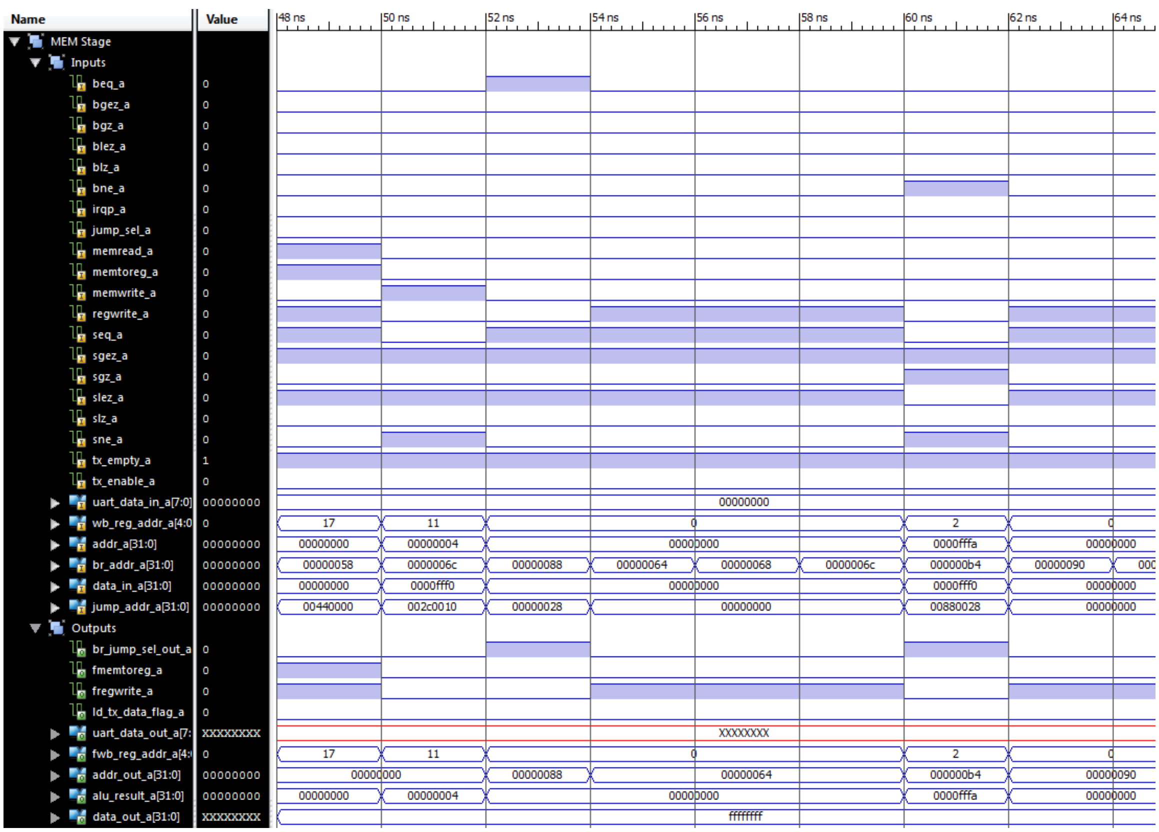
J. MEM STAGE

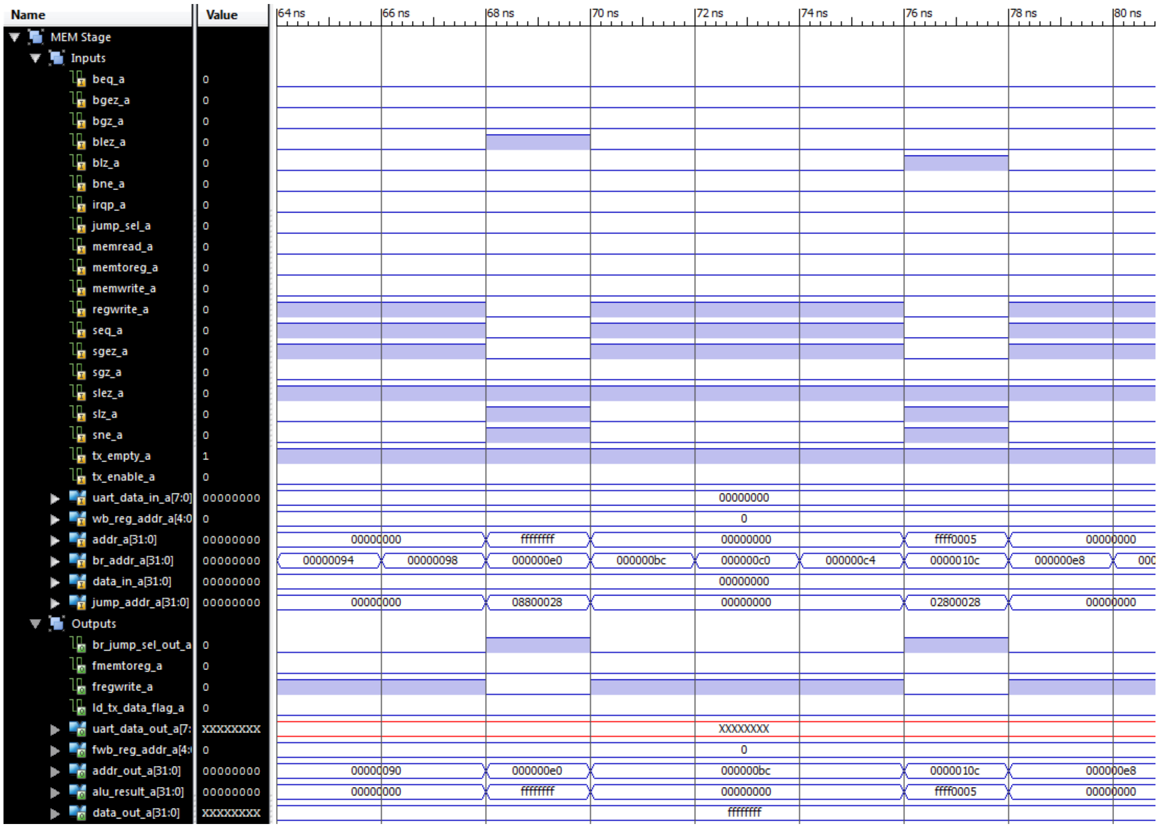


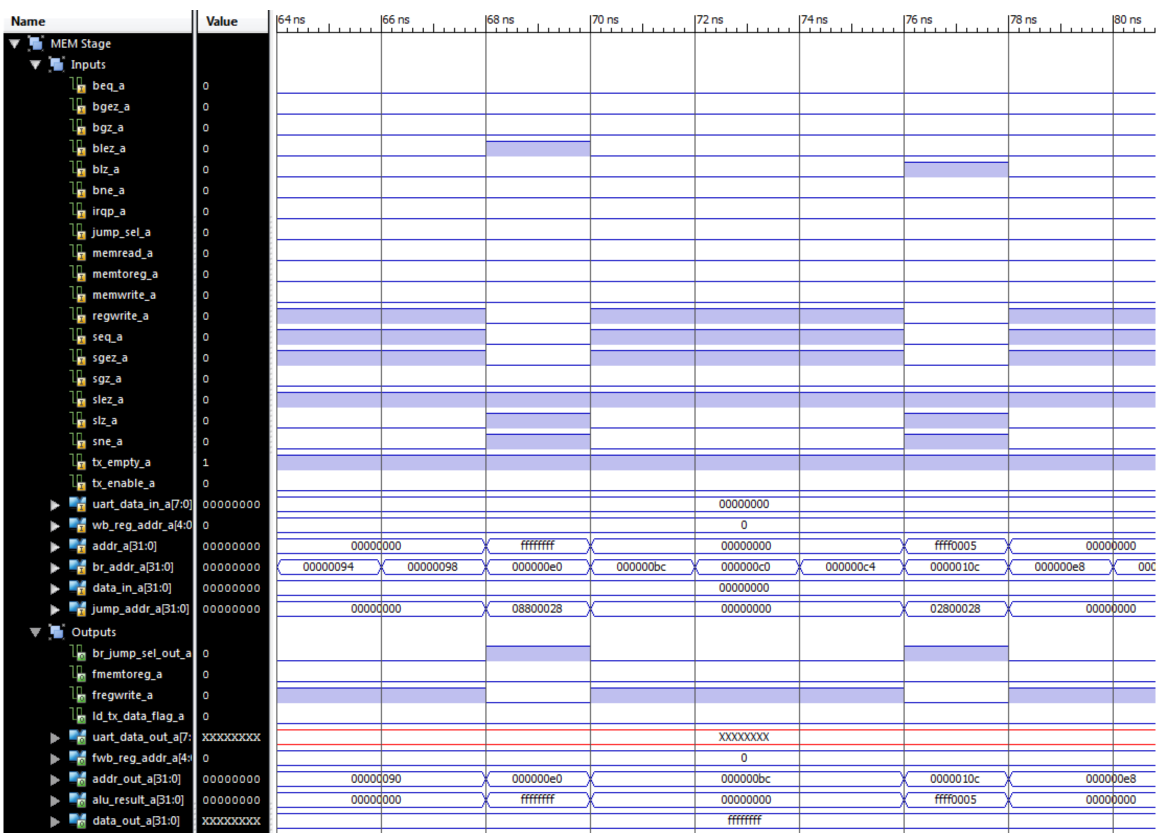


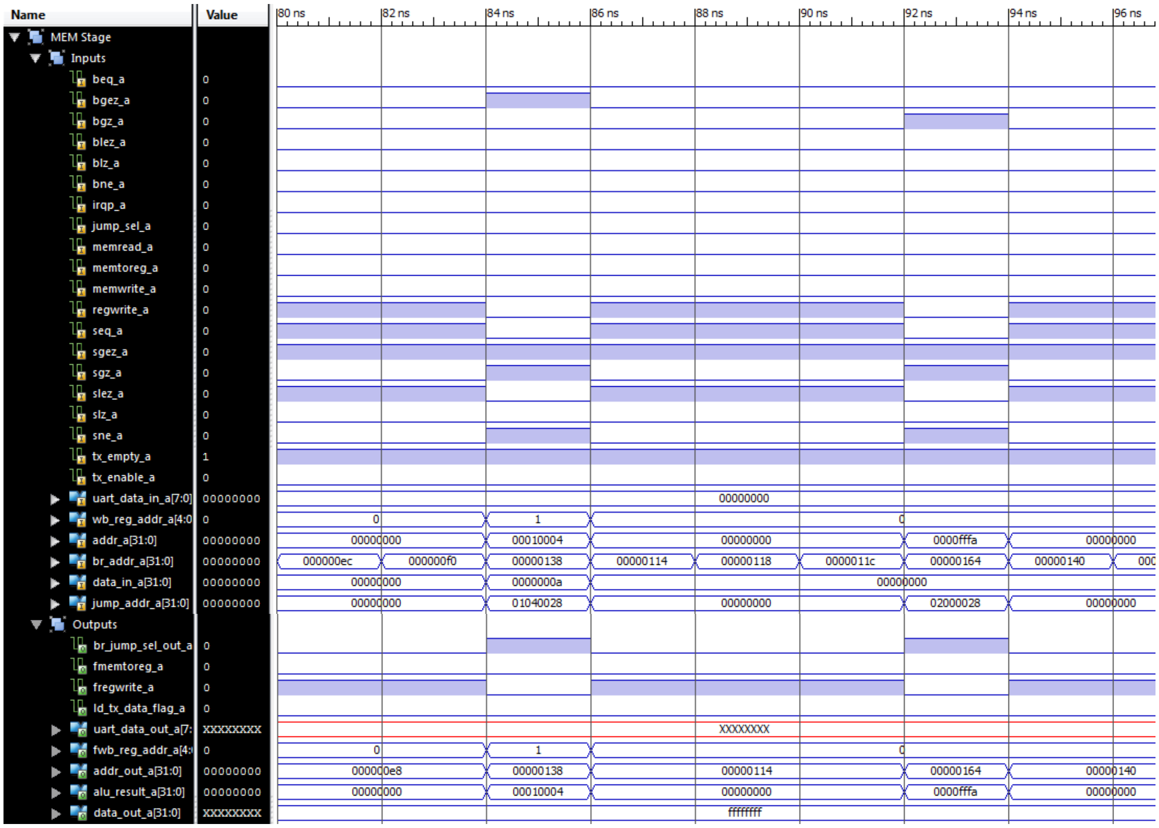


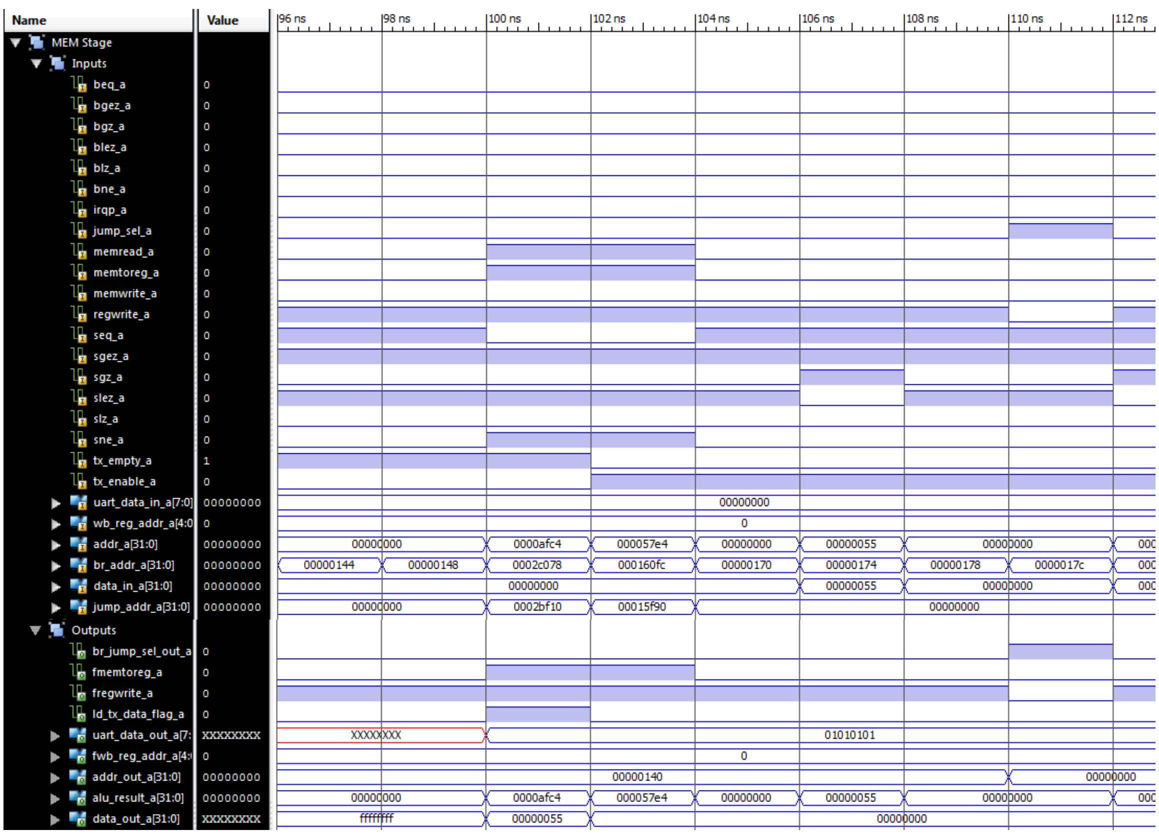

Name
Value
MEM Stage
Inputs
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_sel_a
memread_a
memtoreg_a
memwrite_a
regwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
tx_empty_a
tx_enable_a
uart_data_in_a[7:0]
wb_reg_addr_a[4:0]
addr_a[31:0]
br_addr_a[31:0]
data_in_a[31:0]
jump_addr_a[31:0]
Outputs
br_jump_sel_out_a
fmemtoreg_a
fregwrite_a
ld_tx_data_flag_a
uart_data_out_a[7:
fwb_reg_addr_a[4:
addr_out_a[31:0]
alu_result_a[31:0]
data_out_a[31:0]

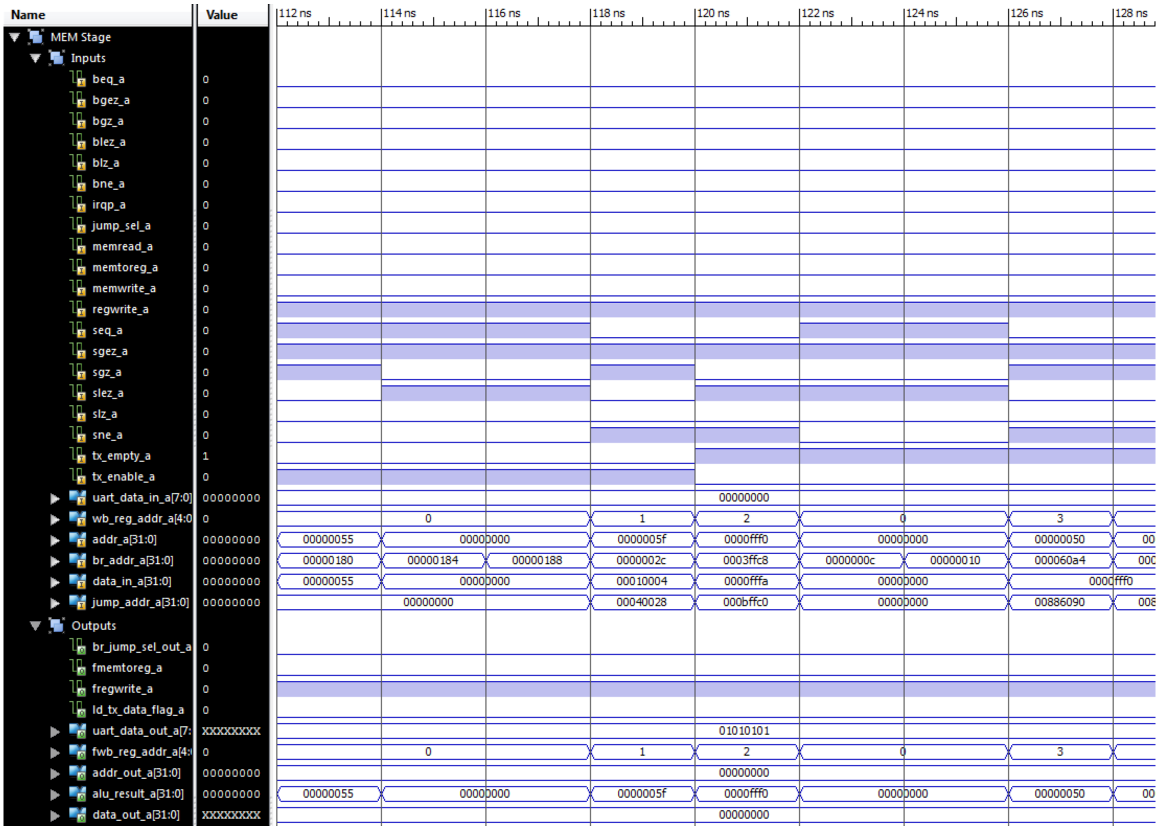

Name
Value
MEM Stage
Inputs
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_sel_a
memread_a
memtoreg_a
memwrite_a
regwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
tx_empty_a
tx_enable_a
uart_data_in_a[7:0]
wb_reg_addr_a[4:0]
addr_a[31:0]
br_addr_a[31:0]
data_in_a[31:0]
jump_addr_a[31:0]
Outputs
br_jump_sel_out_a
fmemtoreg_a
fregwrite_a
ld_tx_data_flag_a
uart_data_out_a[7:
fwb_reg_addr_a[4:
addr_out_a[31:0]
alu_result_a[31:0]
data_out_a[31:0]

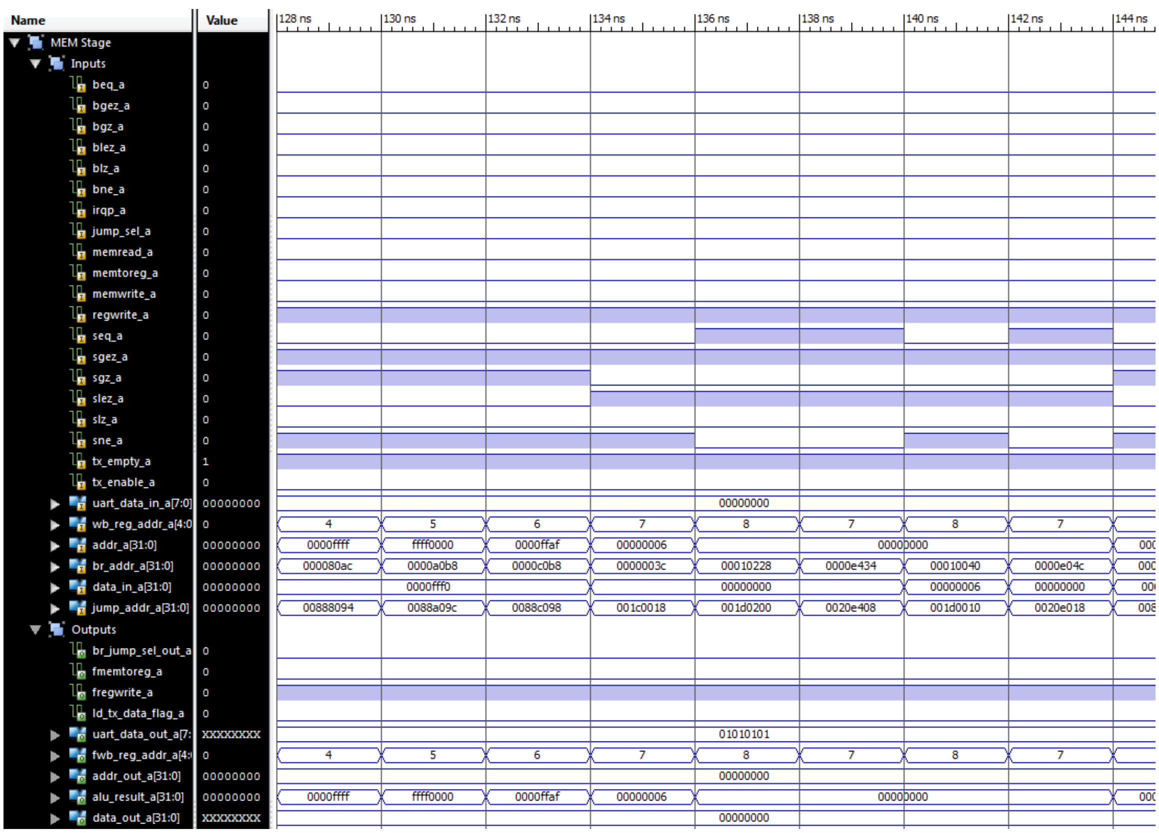

Name
Value
64 ns
66 ns
68 ns
70 ns
72 ns
74 ns
76 ns
78 ns
80 ns
MEM Stage
Inputs
beq_a 0
bgez_a 0
bgz_a 0
blez_a 0
blz_a 0
bne_a 0
irqp_a 0
jump_sel_a 0
memread_a 0
memtoreg_a 0
memwrite_a 0
regwrite_a 0
seq_a 0
sgez_a 0
sgz_a 0
slez_a 0
slz_a 0
sne_a 0
tx_empty_a 1
tx_enable_a 0
uart_data_in_a[7:0] 00000000
wb_reg_addr_a[4:0] 0
addr_a[31:0] 00000000
br_addr_a[31:0] 00000000
data_in_a[31:0] 00000000
jump_addr_a[31:0] 00000000
Outputs
br_jump_sel_out_a 0
fmemtoreg_a 0
fregwrite_a 0
ld_tx_data_flag_a 0
uart_data_out_a[7: XXXXXXXX
fwb_reg_addr_a[4: 0
addr_out_a[31:0] 00000000
alu_result_a[31:0] 00000000
data_out_a[31:0] XXXXXXXX
00000000
0
00000000 ffffffff 00000000 ffff0005 00000000
00000094 00000098 000000e0 000000bc 000000c0 000000c4 0000010c 000000e8 000
00000000
00000000 08800028 00000000 02800028 00000000
XXXXXXXX
0
00000090 000000e0 000000bc 0000010c 000000e8
00000000 ffffffff 00000000 ffff0005 00000000
ffffffff

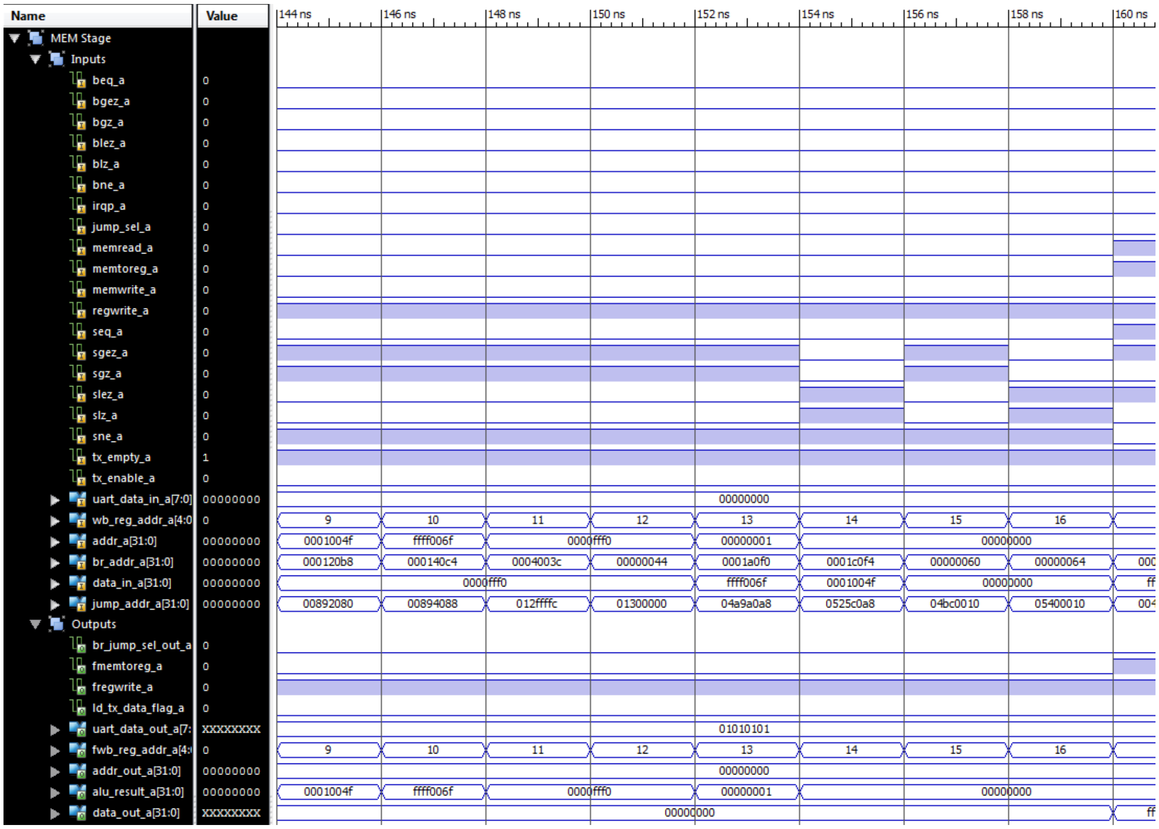


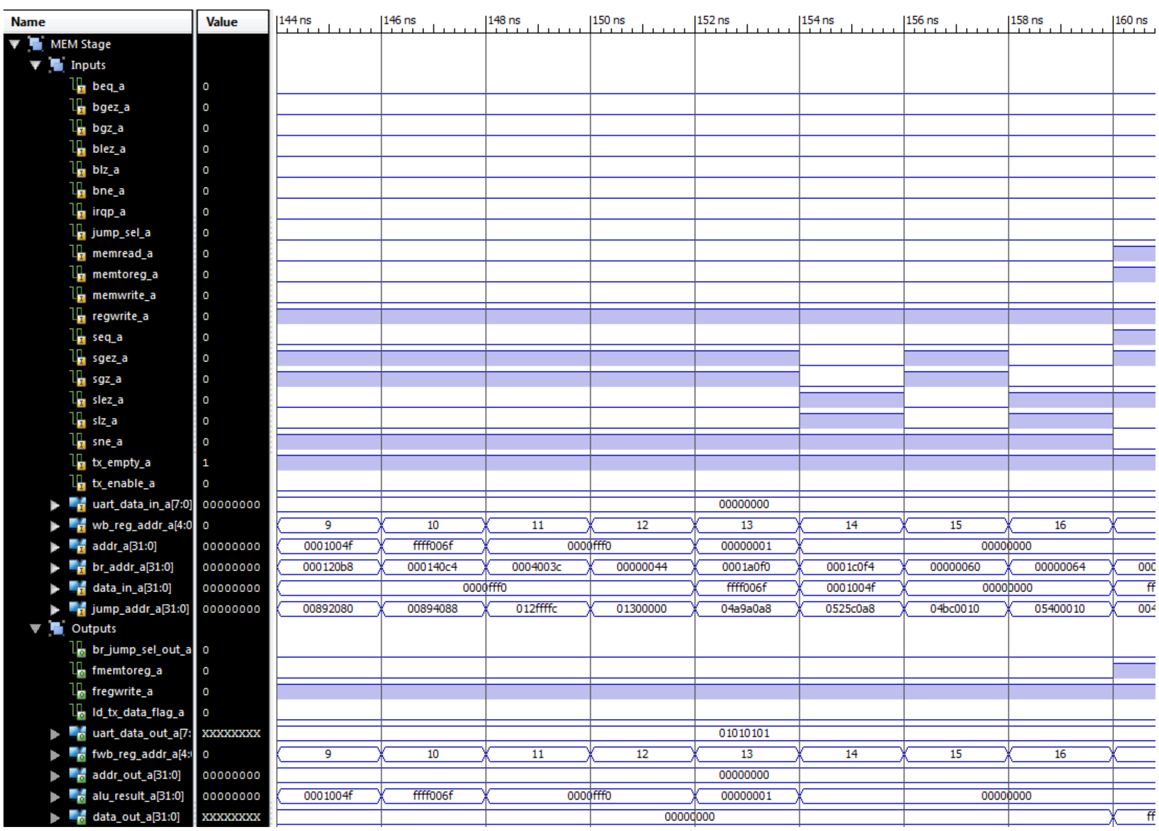

Name
Value
80 ns
82 ns
84 ns
86 ns
88 ns
90 ns
92 ns
94 ns
96 ns
MEM Stage
Inputs
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_sel_a
memread_a
memtoreg_a
memwrite_a
regwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
tx_empty_a
tx_enable_a
uart_data_in_a[7:0]
wb_reg_addr_a[4:0]
addr_a[31:0]
br_addr_a[31:0]
data_in_a[31:0]
jump_addr_a[31:0]
Outputs
br_jump_sel_out_a
fmemtoreg_a
fregwrite_a
ld_tx_data_flag_a
uart_data_out_a[7:
fwb_reg_addr_a[4:
addr_out_a[31:0]
alu_result_a[31:0]
data_out_a[31:0]
00000000
00010004
0000fffa
000000ec
000000f0
00000138
00000114
00000118
0000011c
00000164
00000140
0000000a
01040028
02000028
XXXXXXXX
000000e8
ffffffff

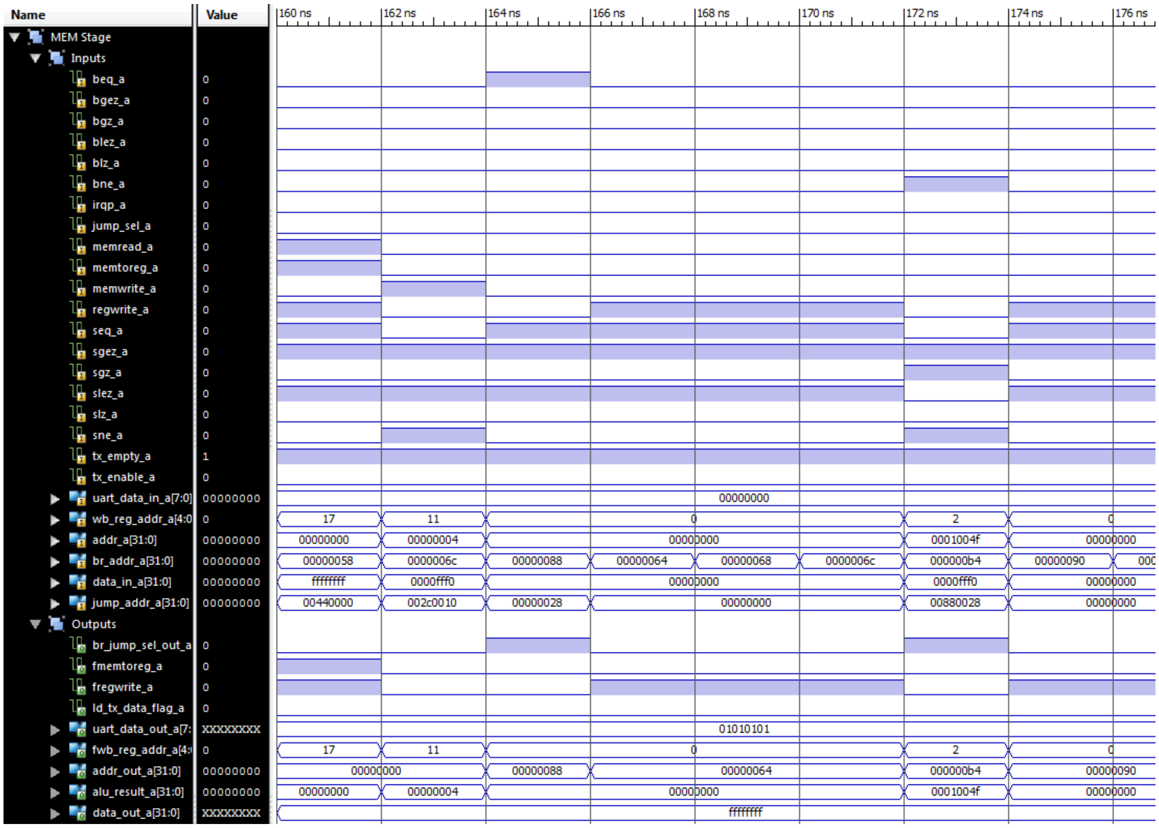

Name
Value
96 ns
98 ns
100 ns
102 ns
104 ns
106 ns
108 ns
110 ns
112 ns
MEM Stage
Inputs
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_sel_a
memread_a
memtoreg_a
memwrite_a
regwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
tx_empty_a
tx_enable_a
uart_data_in_a[7:0]
wb_reg_addr_a[4:0]
addr_a[31:0]
br_addr_a[31:0]
data_in_a[31:0]
jump_addr_a[31:0]
Outputs
br_jump_sel_out_a
fmemtoreg_a
fregwrite_a
ld_tx_data_flag_a
uart_data_out_a[7:
fwb_reg_addr_a[4:
addr_out_a[31:0]
alu_result_a[31:0]
data_out_a[31:0]
00000000
0000afc4
000057e4
00000055
00000144
00000148
0002c078
000160fc
00000170
00000174
00000178
0000017c
0002bf10
00015f90
XXXXXXXX
01010101
00000140
ffffffff

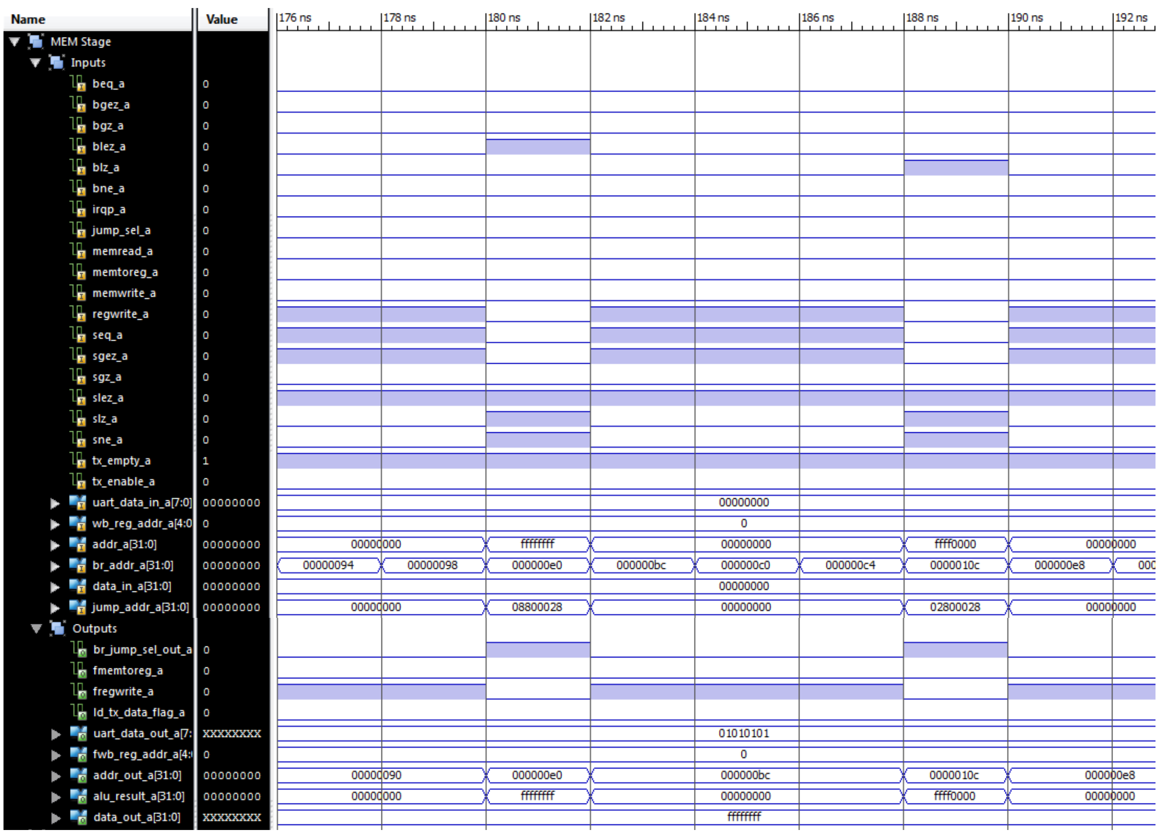


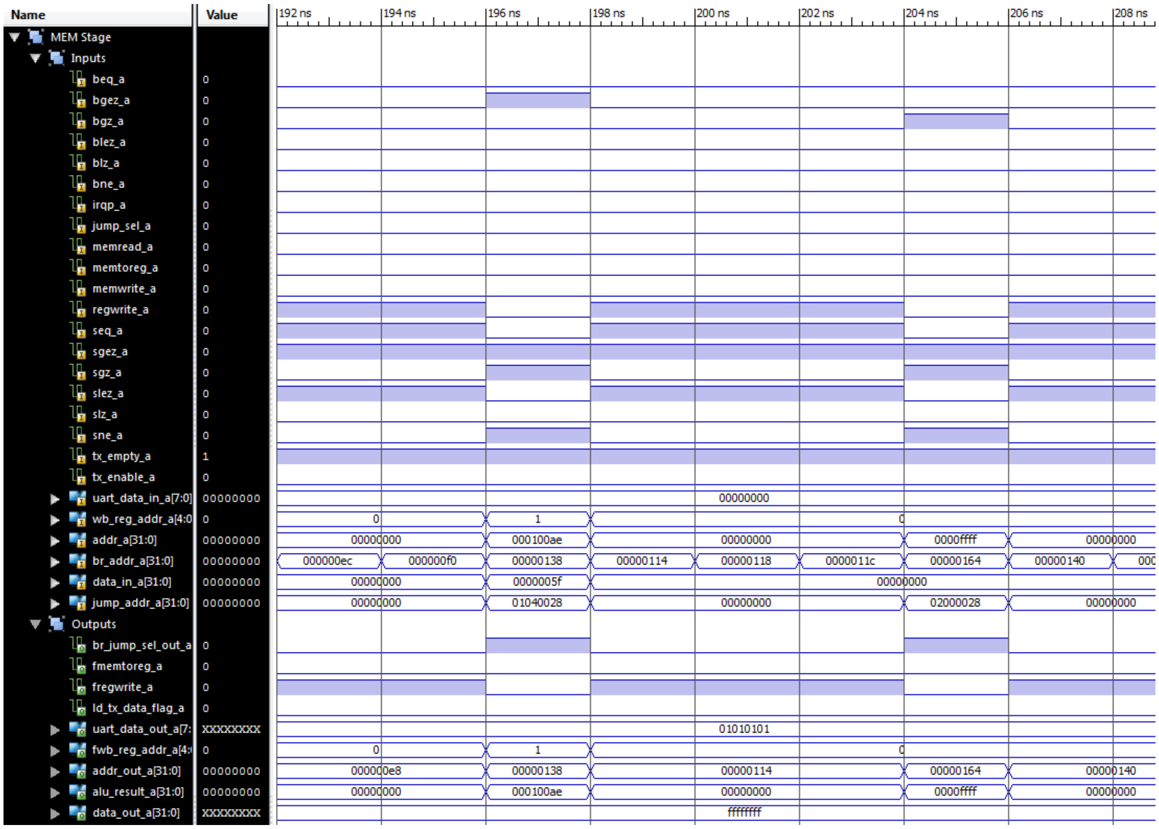


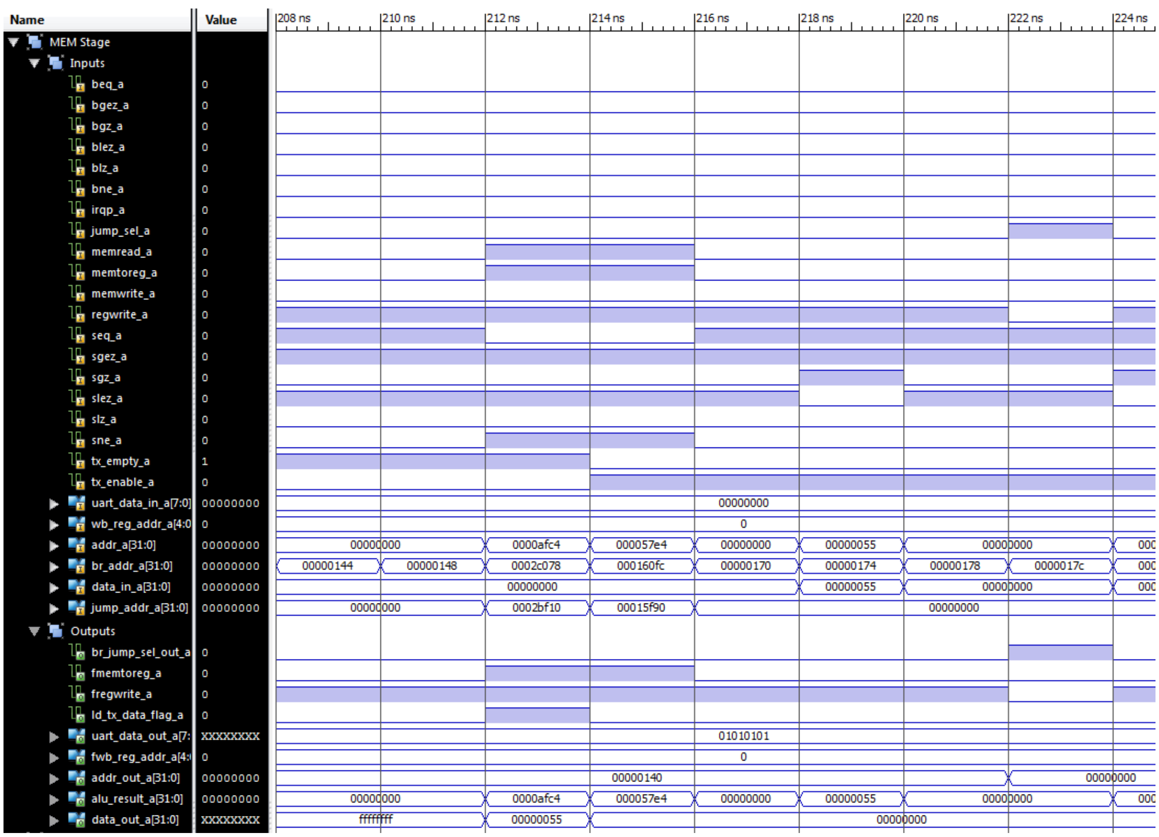

MEM Stage
Inputs
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_sel_a
memread_a
memtoreg_a
memwrite_a
regwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
tx_empty_a
tx_enable_a
uart_data_in_a[7:0]
wb_reg_addr_a[4:0]
addr_a[31:0]
br_addr_a[31:0]
data_in_a[31:0]
jump_addr_a[31:0]
Outputs
br_jump_sel_out_a
fmemtoreg_a
fregwrite_a
ld_tx_data_flag_a
uart_data_out_a[7:0]
fwb_reg_addr_a[4:0]
addr_out_a[31:0]
alu_result_a[31:0]
data_out_a[31:0]

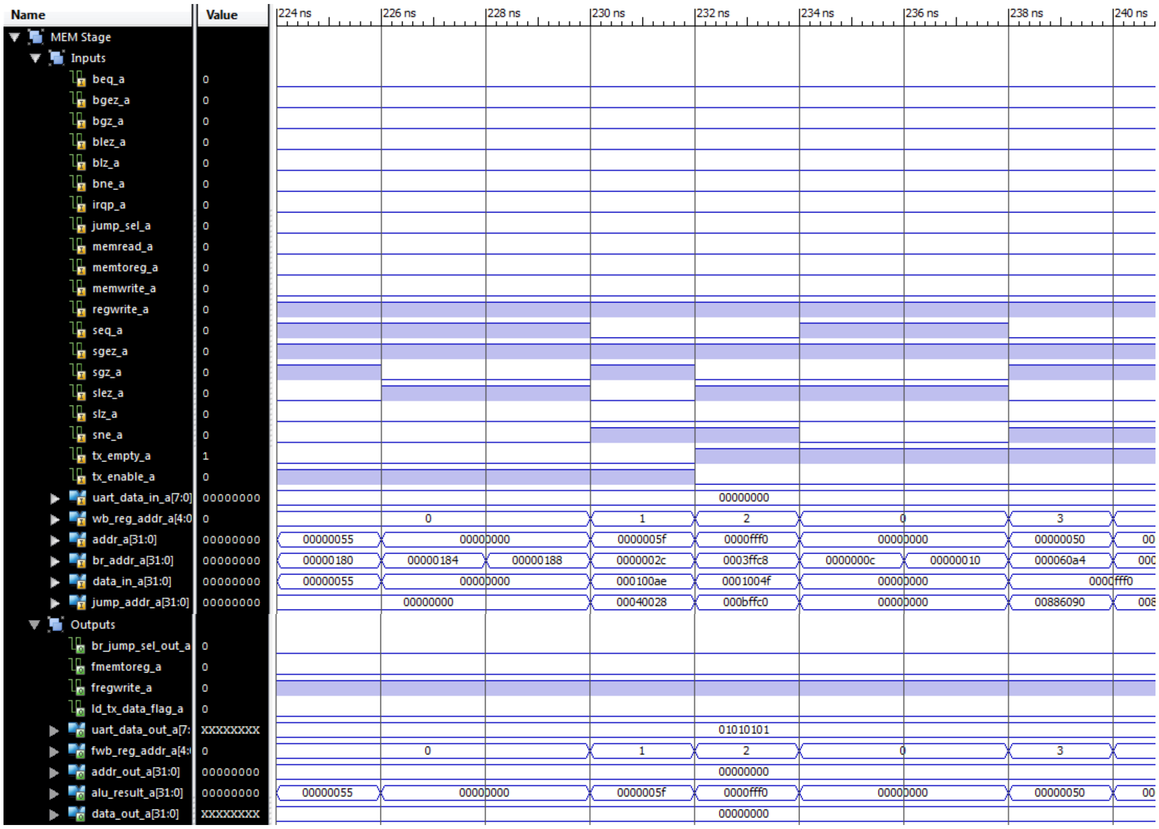

MEM Stage
Inputs
Outputs

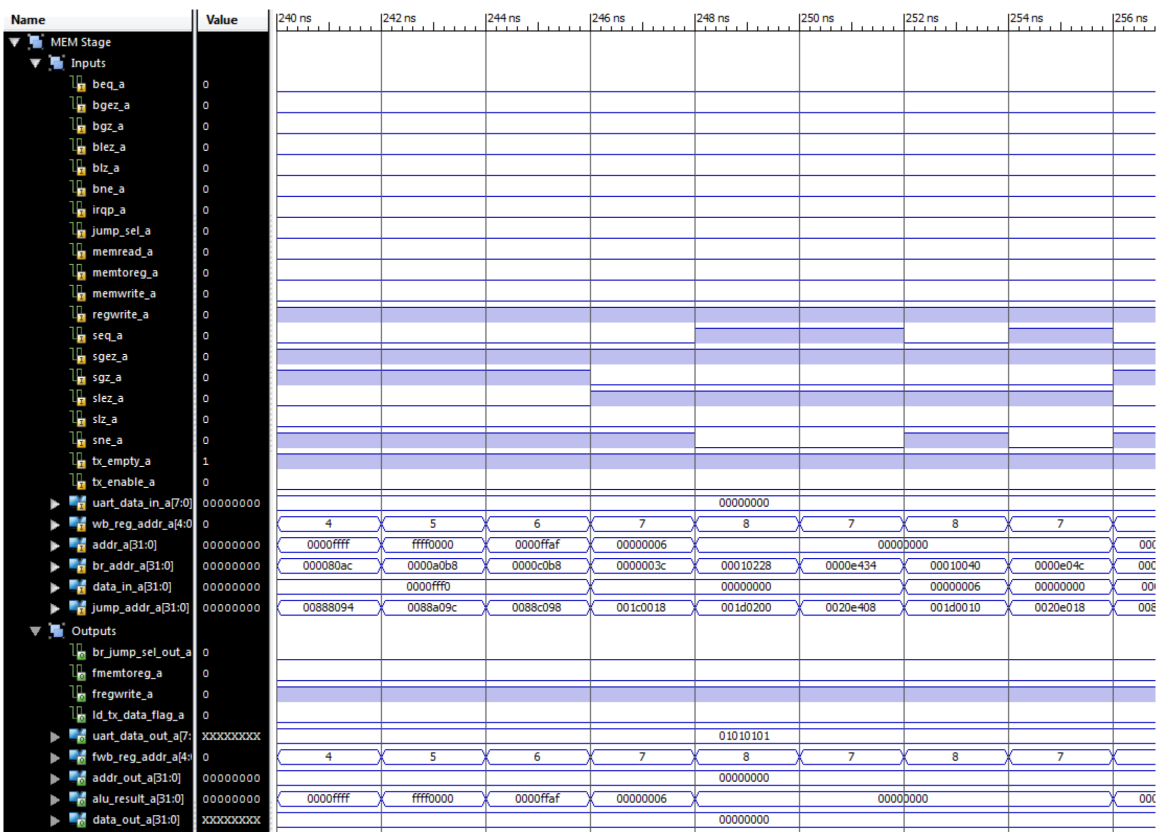


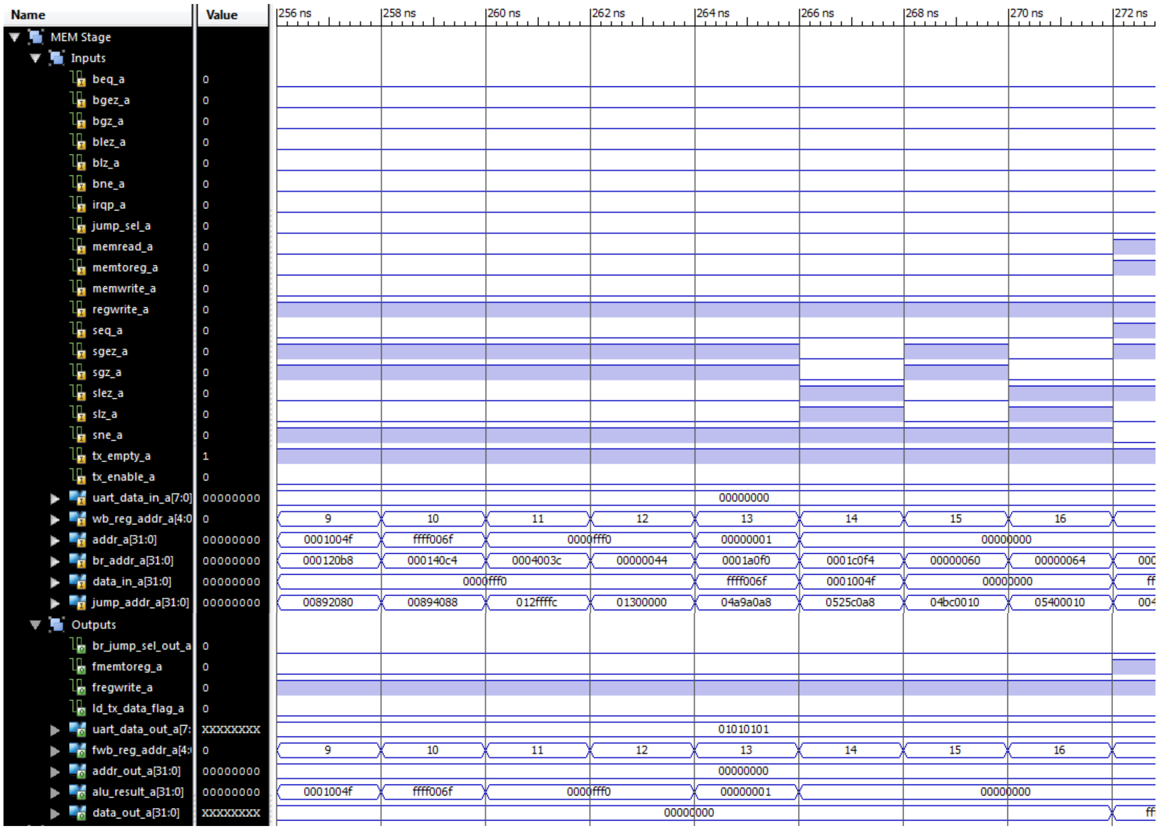


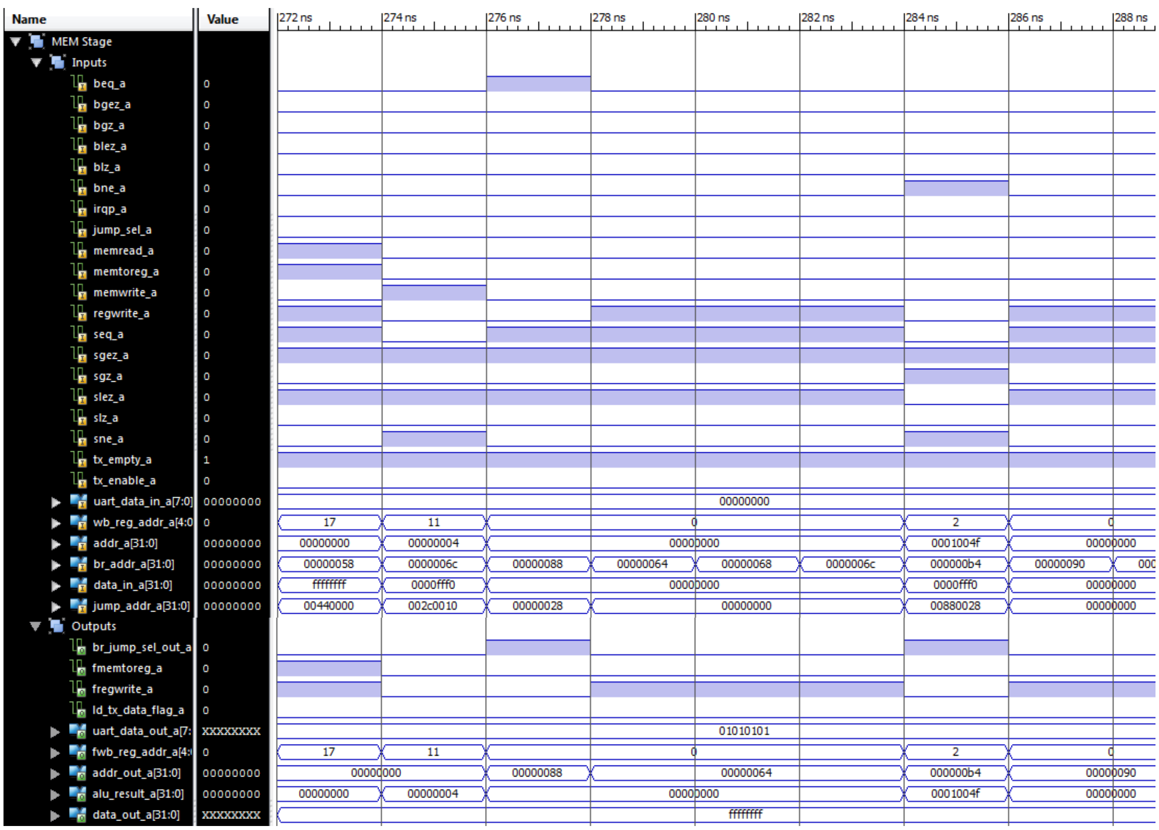

Name
Value
MEM Stage
Inputs
beq_a
bgez_a
bgz_a
blez_a
blz_a
bne_a
irqp_a
jump_sel_a
memread_a
memtoreg_a
memwrite_a
regwrite_a
seq_a
sgez_a
sgz_a
slez_a
slz_a
sne_a
tx_empty_a
tx_enable_a
uart_data_in_a[7:0]
wb_reg_addr_a[4:0]
addr_a[31:0]
br_addr_a[31:0]
data_in_a[31:0]
jump_addr_a[31:0]
Outputs
br_jump_sel_out_a
fmemtoreg_a
fregwrite_a
ld_tx_data_flag_a
uart_data_out_a[7:
fwb_reg_addr_a[4:
addr_out_a[31:0]
alu_result_a[31:0]
data_out_a[31:0]
192 ns
194 ns
196 ns
198 ns
200 ns
202 ns
204 ns
206 ns
208 ns

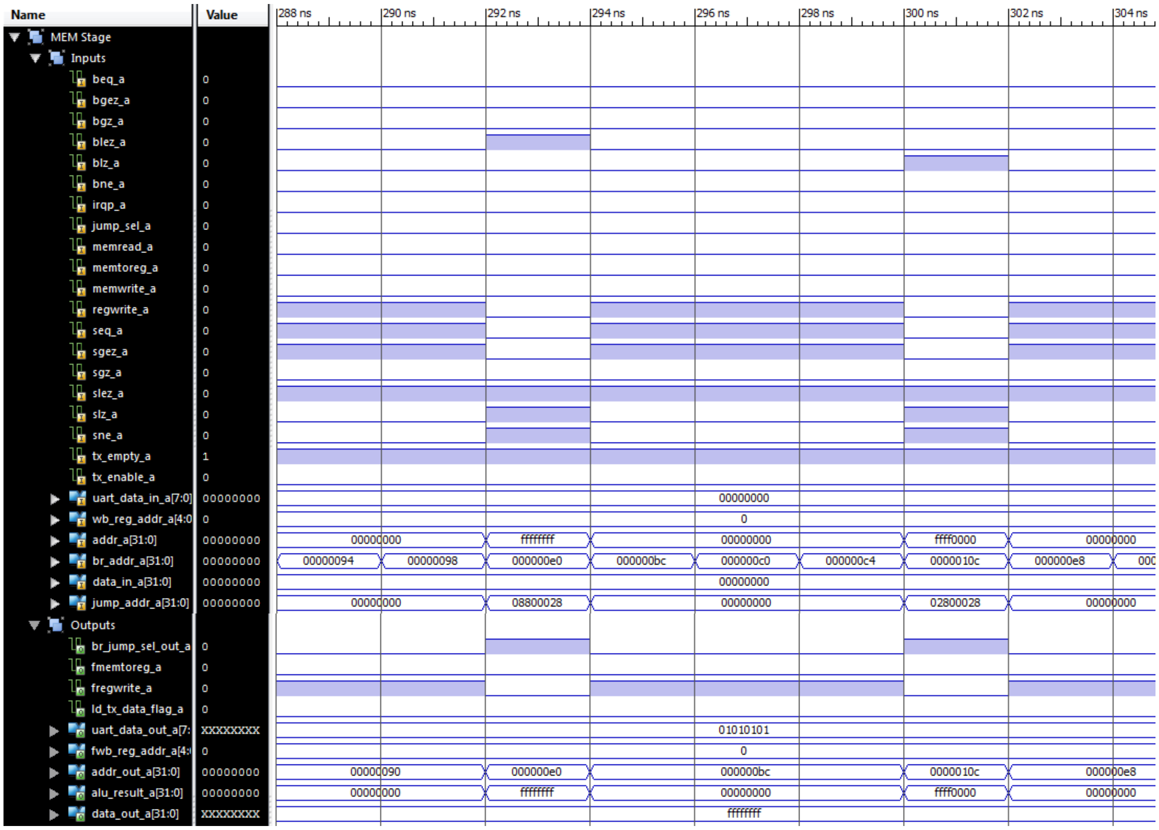

Name
Value
MEM Stage
Inputs
Outputs
208 ns
210 ns
212 ns
214 ns
216 ns
218 ns
220 ns
222 ns
224 ns

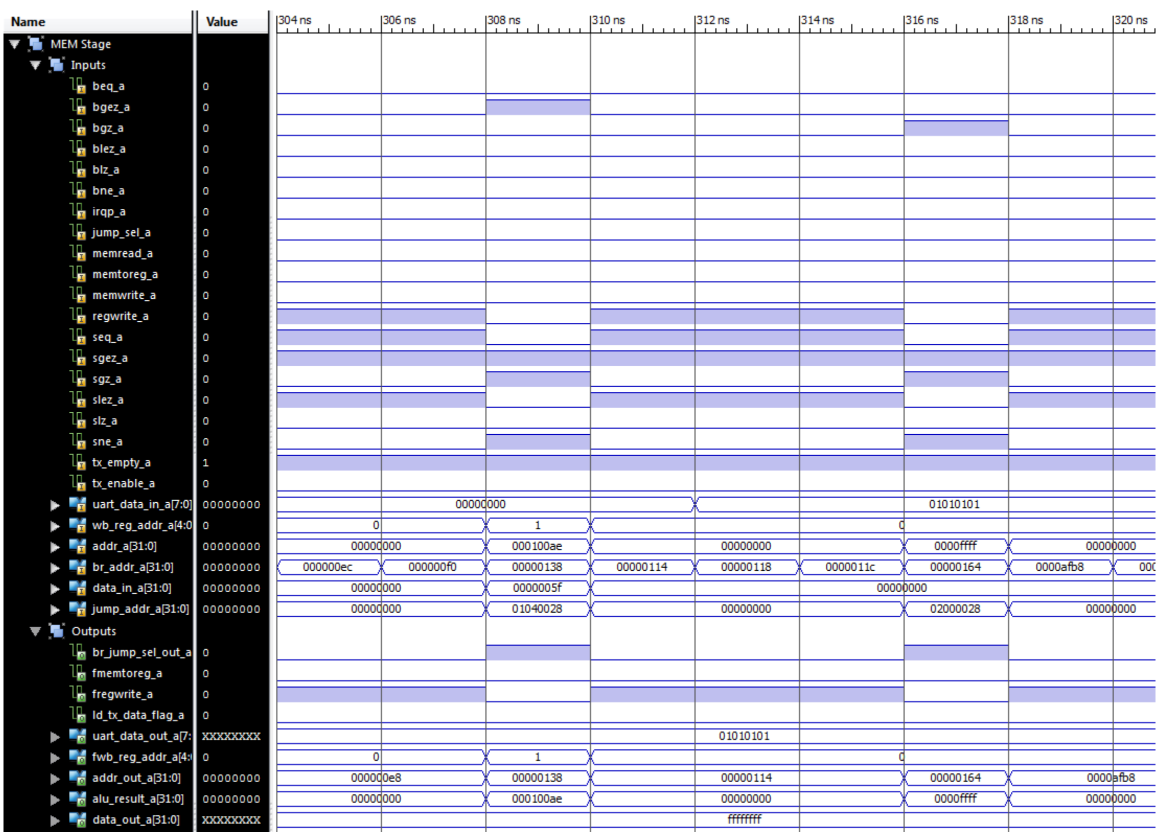


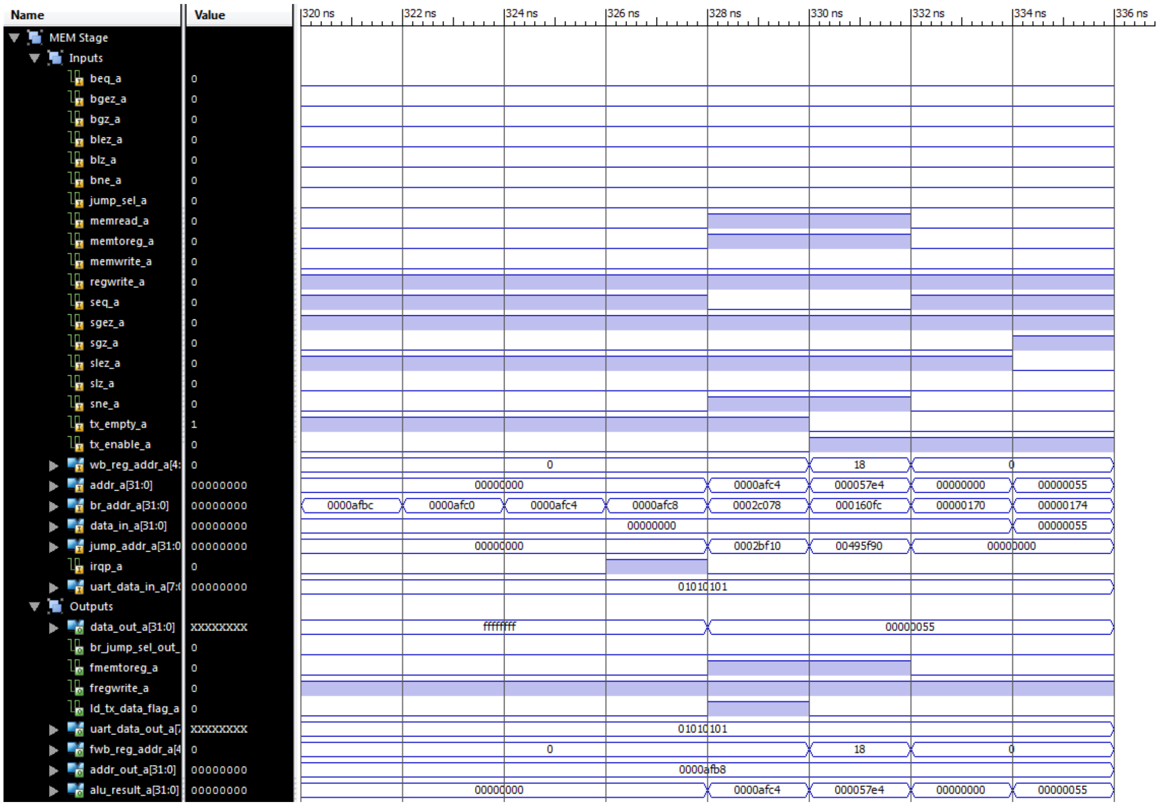




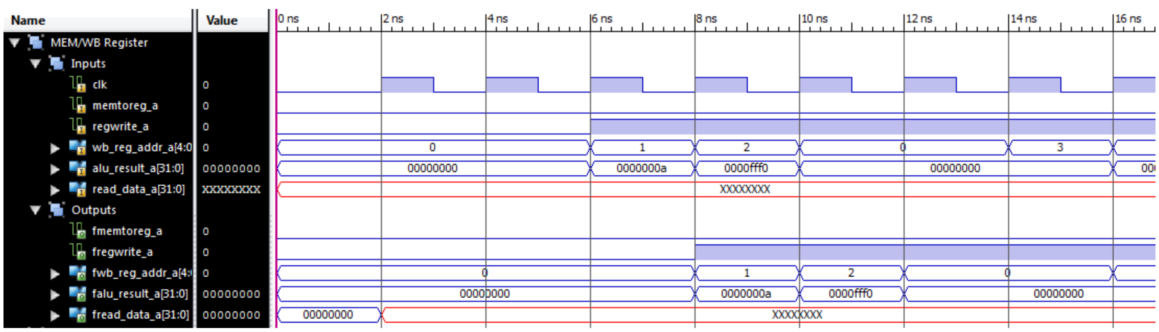


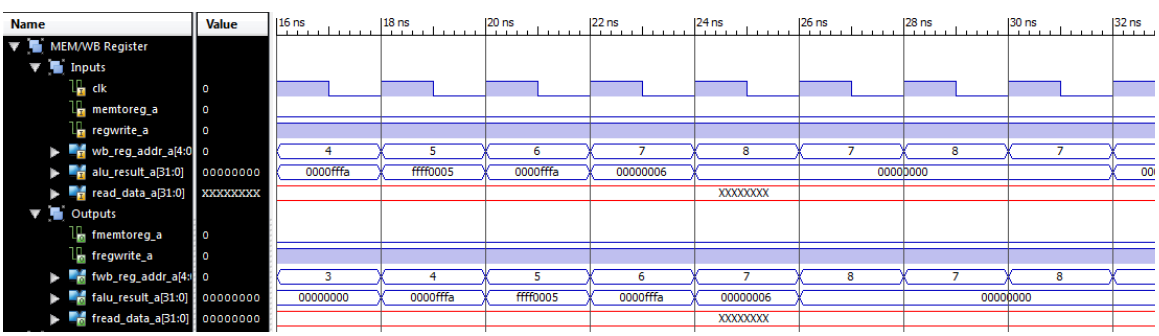

Name
Value
288 ns
290 ns
292 ns
294 ns
296 ns
298 ns
300 ns
302 ns
304 ns
MEM Stage
Inputs
beq_a 0
bgez_a 0
bgz_a 0
blez_a 0
blz_a 0
bne_a 0
irqp_a 0
jump_sel_a 0
memread_a 0
memtoreg_a 0
memwrite_a 0
regwrite_a 0
seq_a 0
sgez_a 0
sgz_a 0
slez_a 0
slz_a 0
sne_a 0
tx_empty_a 1
tx_enable_a 0
uart_data_in_a[7:0] 00000000
wb_reg_addr_a[4:0] 0
addr_a[31:0] 00000000
br_addr_a[31:0] 00000000
data_in_a[31:0] 00000000
jump_addr_a[31:0] 00000000
Outputs
br_jump_sel_out_a 0
fmemtoreg_a 0
fregwrite_a 0
ld_tx_data_flag_a 0
uart_data_out_a[7: XXXXXXXX
fwb_reg_addr_a[4: 0
addr_out_a[31:0] 00000000
alu_result_a[31:0] 00000000
data_out_a[31:0] XXXXXXXX
00000000
0
00000000 ffffffff 00000000 ffff0000 00000000
00000094 00000098 000000e0 000000bc 000000c0 000000c4 0000010c 000000e8
00000000
00000000 08800028 00000000 02800028 00000000
01010101
0
00000090 000000e0 000000bc 0000010c 000000e8
00000000 ffffffff 00000000 ffff0000 00000000
ffffffff

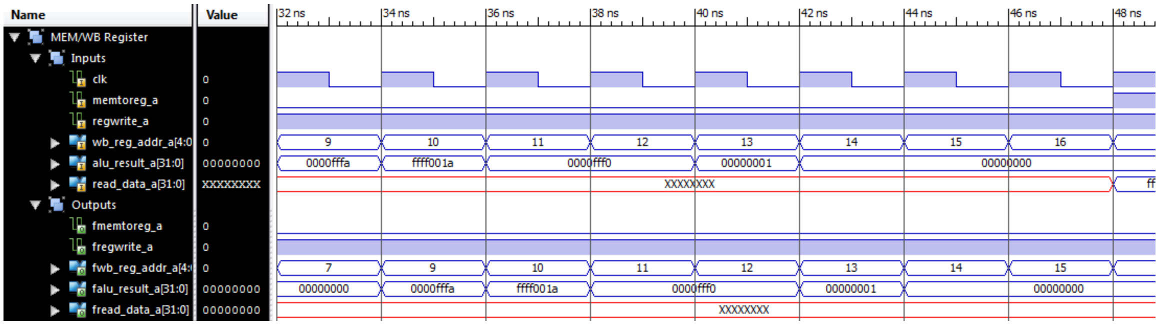

Name
Value
304 ns
306 ns
308 ns
310 ns
312 ns
314 ns
316 ns
318 ns
320 ns
MEM Stage
Inputs
beq_a 0
bgez_a 0
bgz_a 0
blez_a 0
blz_a 0
bne_a 0
irqp_a 0
jump_sel_a 0
memread_a 0
memtoreg_a 0
memwrite_a 0
regwrite_a 0
seq_a 0
sgez_a 0
sgz_a 0
slez_a 0
slz_a 0
sne_a 0
tx_empty_a 1
tx_enable_a 0
uart_data_in_a[7:0] 00000000
wb_reg_addr_a[4:0] 0
addr_a[31:0] 00000000
br_addr_a[31:0] 00000000
data_in_a[31:0] 00000000
jump_addr_a[31:0] 00000000
Outputs
br_jump_sel_out_a 0
fmemtoreg_a 0
fregwrite_a 0
ld_tx_data_flag_a 0
uart_data_out_a[7: XXXXXXXX
fwb_reg_addr_a[4: 0
addr_out_a[31:0] 00000000
alu_result_a[31:0] 00000000
data_out_a[31:0] XXXXXXXX
00000000 01010101
0 1 0
00000000 000100ae 00000000 0000ffff 00000000
000000ec 000000f0 00000138 00000114 00000118 0000011c 00000164 0000afb8
00000000 0000005f 00000000
00000000 01040028 00000000 02000028 00000000
01010101
0 1 0
000000e8 00000138 00000114 00000164 0000afb8
00000000 000100ae 00000000 0000ffff 00000000
ffffffff

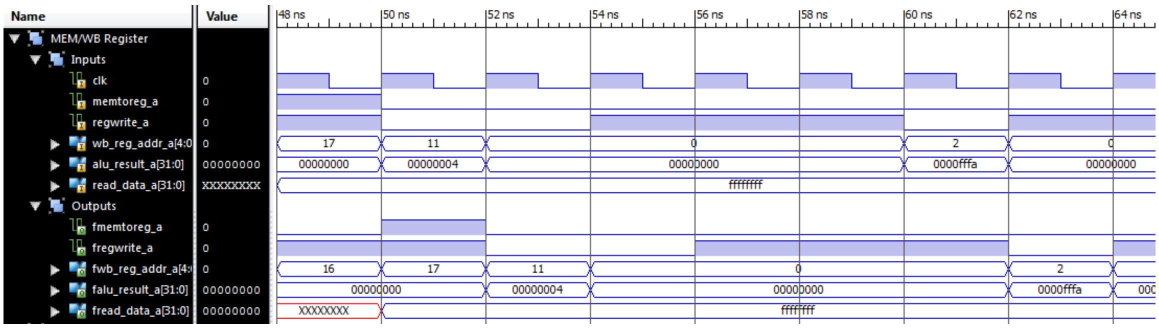


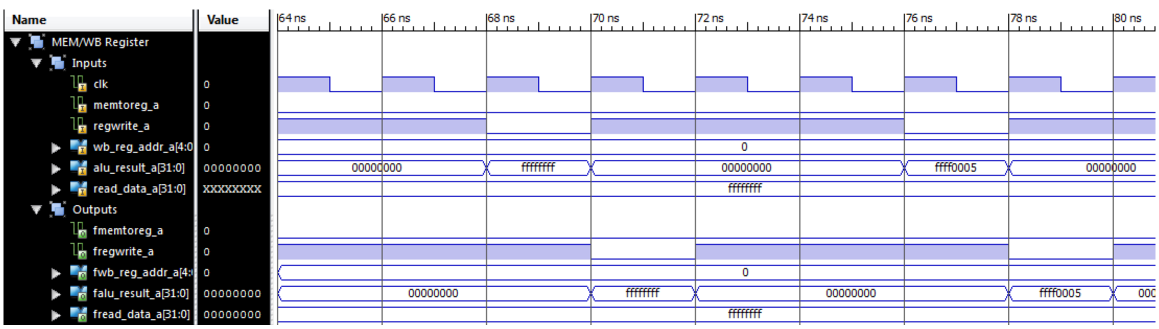
K. MEM/WB REGISTER

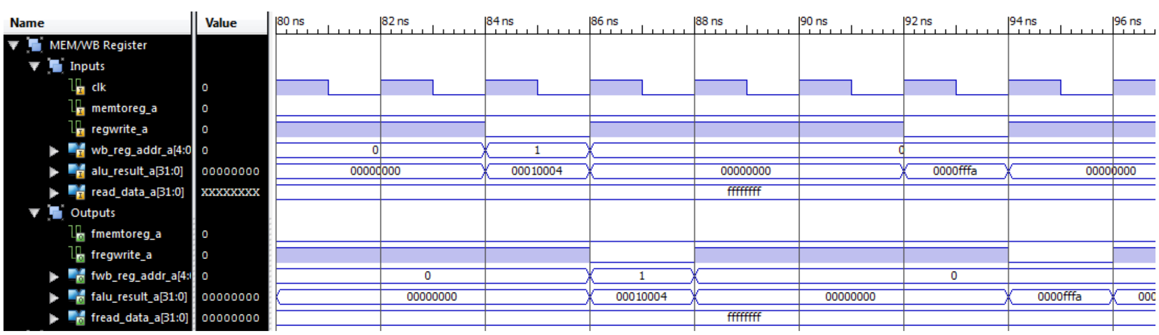


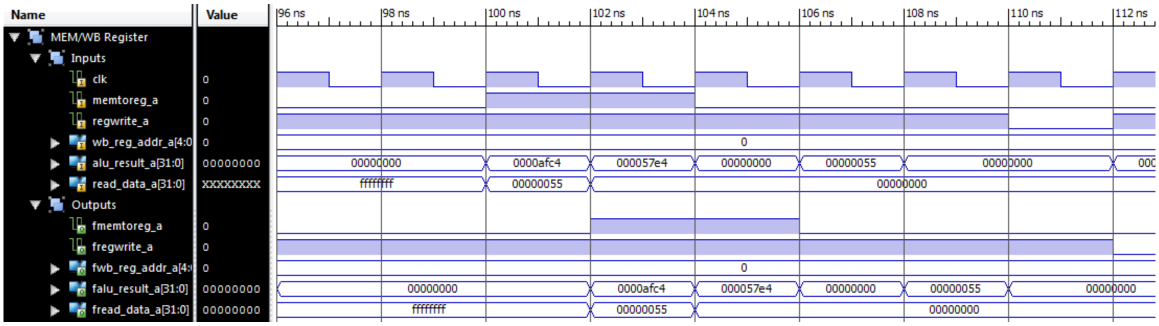



Name
Value
32 ns
34 ns
36 ns
38 ns
40 ns
42 ns
44 ns
46 ns
48 ns
MEM/WB Register
Inputs
clk 0
memtoreg_a 0
regwrite_a 0
wb_reg_addr_a[4:0] 0
alu_result_a[31:0] 00000000
read_data_a[31:0] XXXXXXXX
Outputs
fmemtoreg_a 0
fregwrite_a 0
fwb_reg_addr_a[4:0] 0
falu_result_a[31:0] 00000000
fread_data_a[31:0] 00000000
9
10
11
12
13
14
15
16
0000fffa
ffff001a
0000fff0
00000001
00000000
XXXXXXXX
ff
7
9
10
11
12
13
14
15
00000000
0000fffa
ffff001a
0000fff0
00000001
00000000
XXXXXXXX

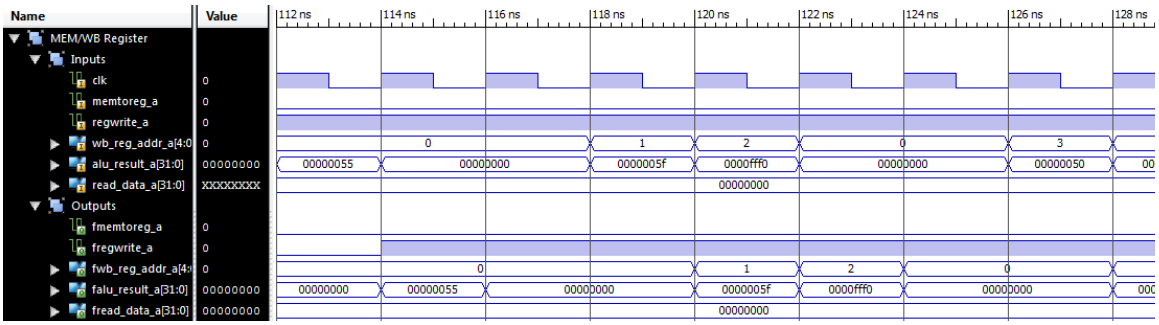

Name
Value
48 ns
50 ns
52 ns
54 ns
56 ns
58 ns
60 ns
62 ns
64 ns
MEM/WB Register
Inputs
clk 0
memtoreg_a 0
regwrite_a 0
wb_reg_addr_a[4:0] 0
alu_result_a[31:0] 00000000
read_data_a[31:0] XXXXXXXX
Outputs
fmemtoreg_a 0
fregwrite_a 0
fwb_reg_addr_a[4:0] 0
falu_result_a[31:0] 00000000
fread_data_a[31:0] 00000000
17
11
0
2
0
00000000
00000004
00000000
0000fffa
00000000
ffffffff
16
17
11
0
2
00000000
00000004
00000000
0000fffa
000
XXXXXXXX
ffffffff


Name
Value
64 ns
66 ns
68 ns
70 ns
72 ns
74 ns
76 ns
78 ns
80 ns
MEM/WB Register
Inputs
clk 0
memtoreg_a 0
regwrite_a 0
wb_reg_addr_a[4:0] 0
alu_result_a[31:0] 00000000
read_data_a[31:0] XXXXXXXX
Outputs
fmemtoreg_a 0
fregwrite_a 0
fwb_reg_addr_a[4:0] 0
falu_result_a[31:0] 00000000
fread_data_a[31:0] 00000000
0
00000000
ffffffff
00000000
ffff0005
00000000
ffffffff
0
00000000
ffffffff
00000000
ffff0005
000
ffffffff

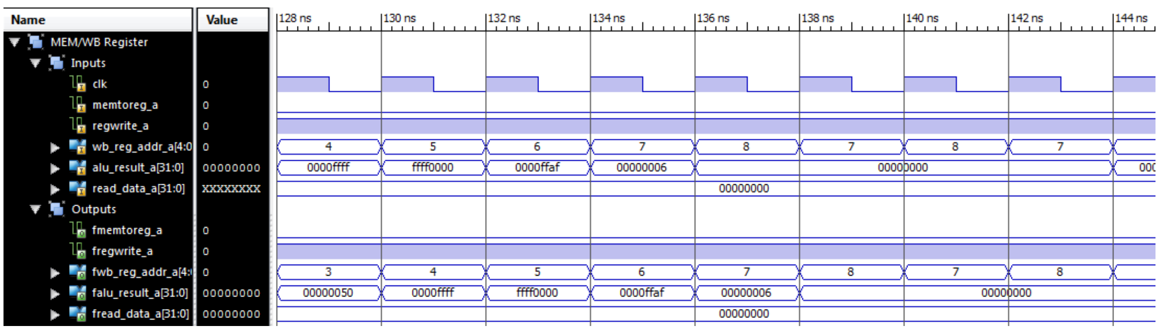

Name
Value
80 ns
82 ns
84 ns
86 ns
88 ns
90 ns
92 ns
94 ns
96 ns
MEM/WB Register
Inputs
clk 0
memtoreg_a 0
regwrite_a 0
wb_reg_addr_a[4:0] 0
alu_result_a[31:0] 00000000
read_data_a[31:0] XXXXXXXX
Outputs
fmemtoreg_a 0
fregwrite_a 0
fwb_reg_addr_a[4:0] 0
falu_result_a[31:0] 00000000
fread_data_a[31:0] 00000000
0
1
0
00000000
00010004
00000000
0000fffa
00000000
ffffffff
0
1
0
00000000
00010004
00000000
0000fffa
000
ffffffff



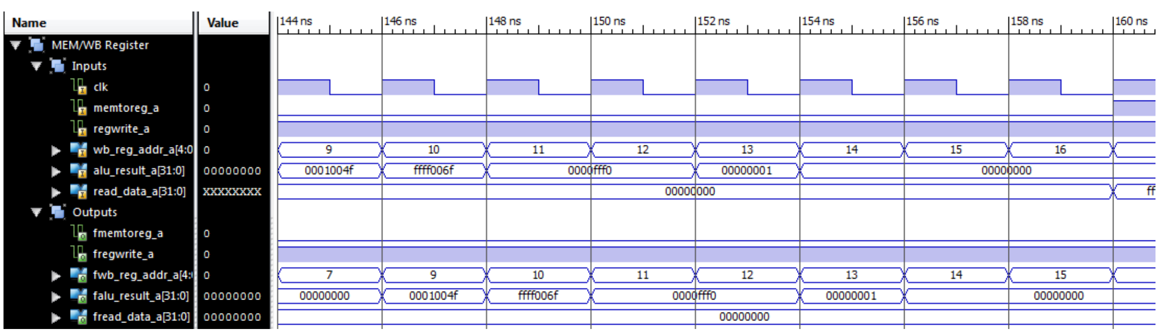
Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
0
0
0
0
00000000
XXXXXXXX
0
0
0
00000000
00000000
96 ns
98 ns
100 ns
102 ns
104 ns
106 ns
108 ns
110 ns
112 ns
00000000
0000afc4
000057e4
00000055
ffffffff
00000055



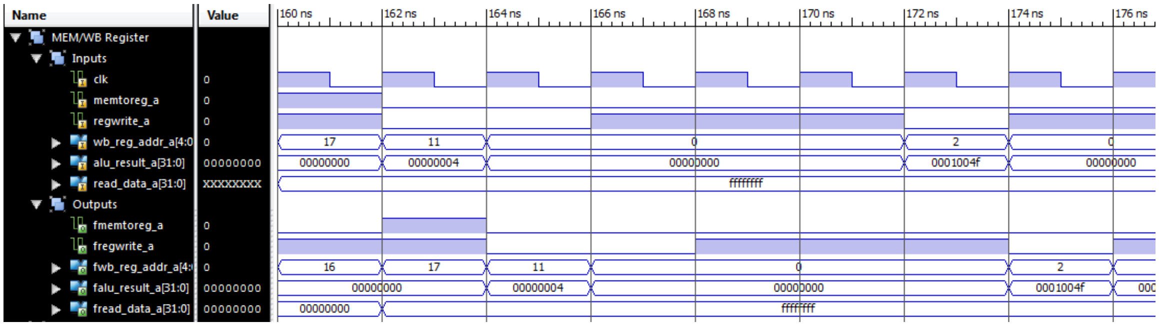
Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
112 ns
114 ns
116 ns
118 ns
120 ns
122 ns
124 ns
126 ns
128 ns
0
1
2
3
00000055
00000000
0000005f
0000fff0
00000050

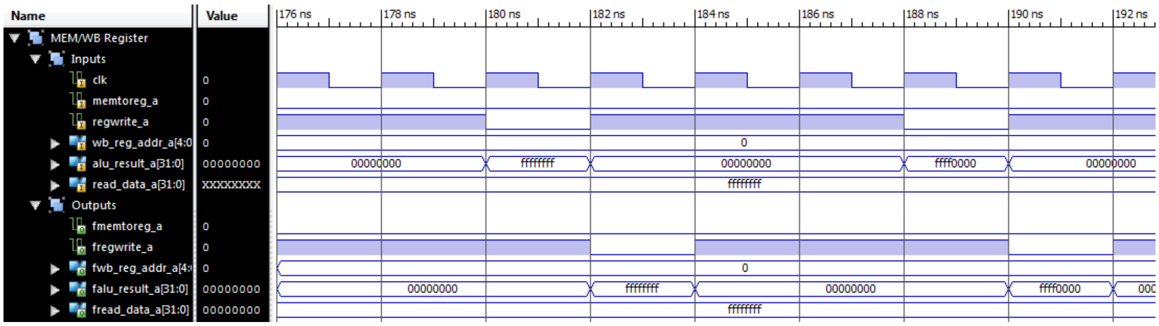


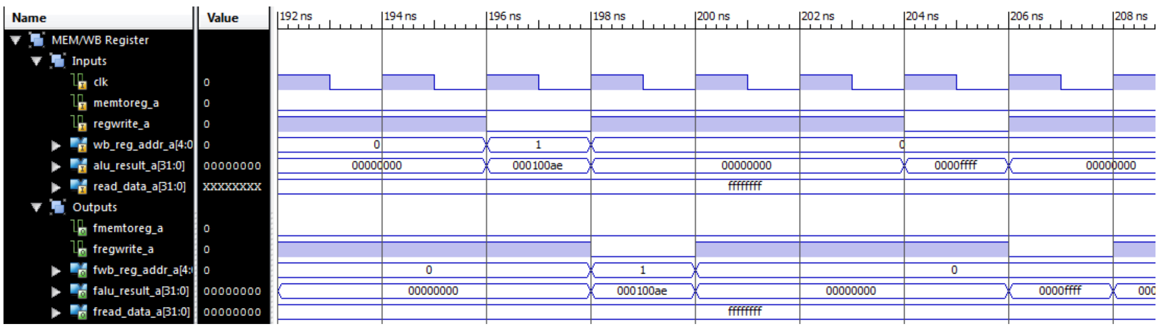
Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
128 ns
130 ns
132 ns
134 ns
136 ns
138 ns
140 ns
142 ns
144 ns
0000ffff
ffff0000
0000ffaf
00000006
00000000
00000050

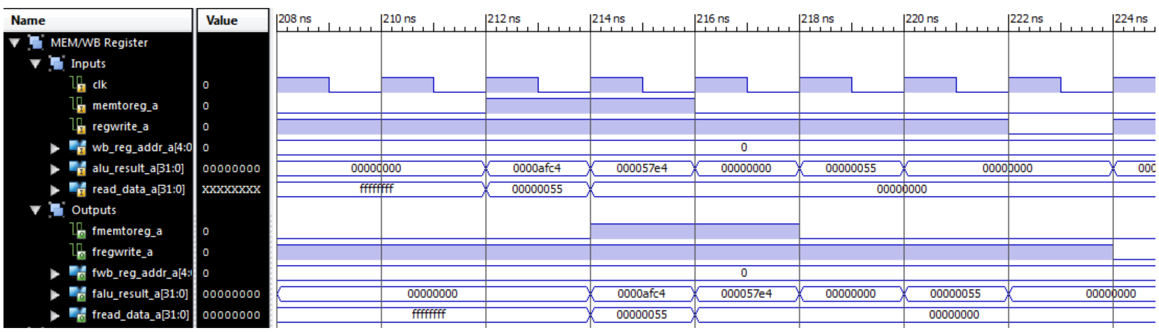


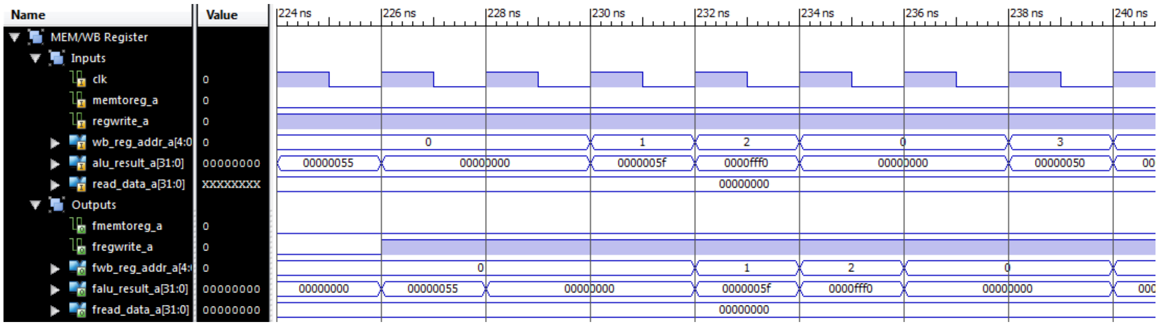
Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
144 ns
146 ns
148 ns
150 ns
152 ns
154 ns
156 ns
158 ns
160 ns
0001004f
ffff006f
0000fff0
00000001
00000000

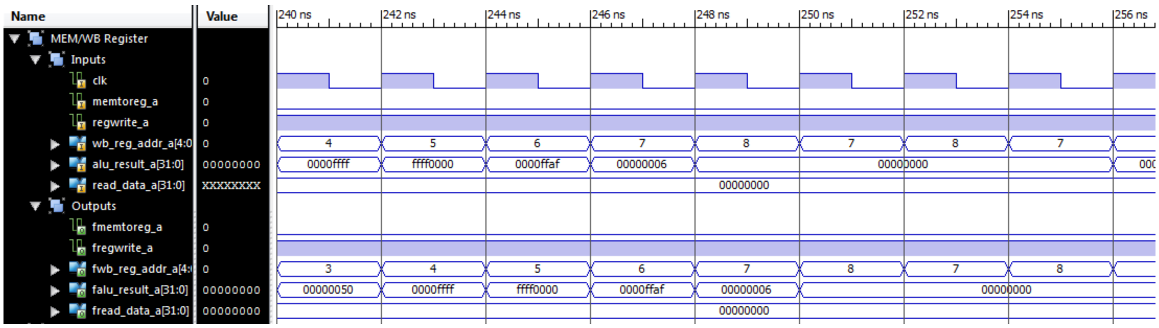


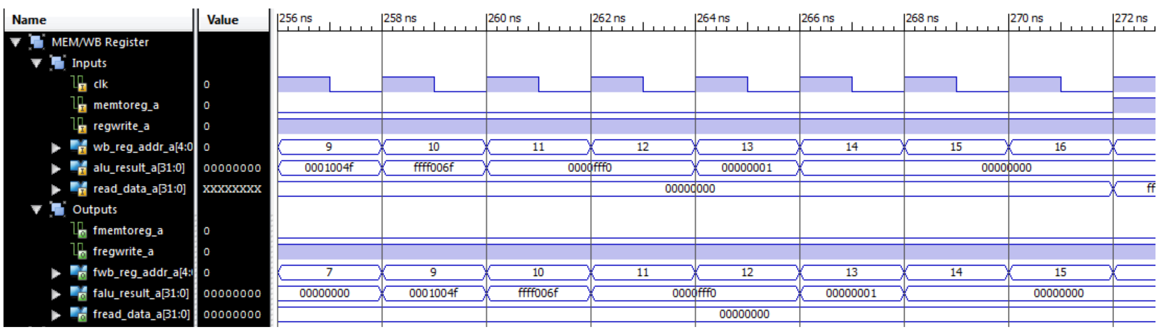


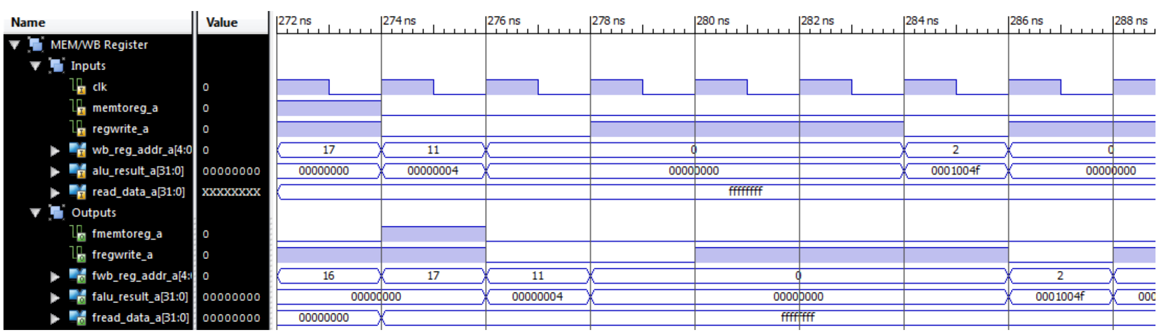


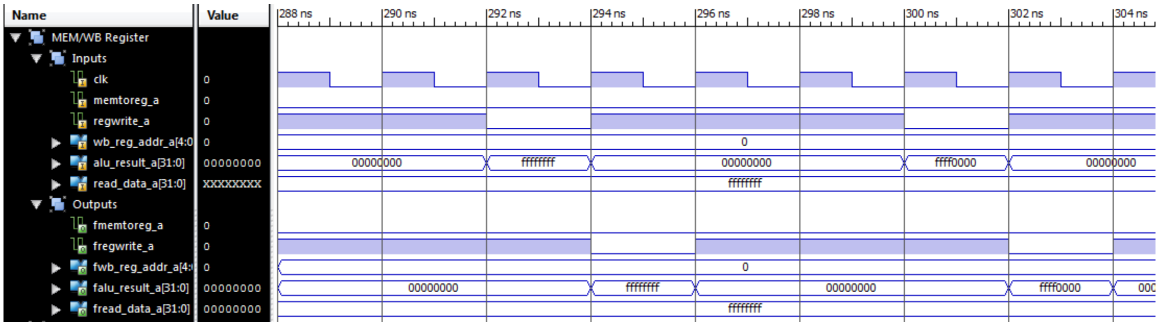


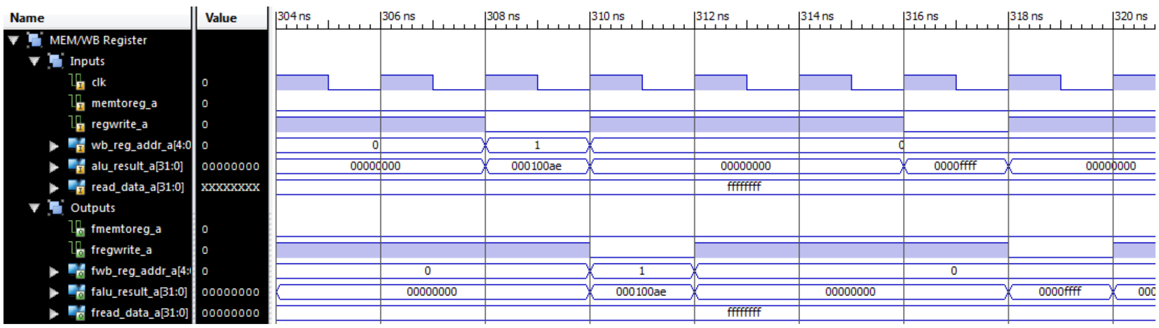

Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
0
0
0
0
00000000
XXXXXXXX
0
0
0
00000000
00000000
224 ns
226 ns
228 ns
230 ns
232 ns
234 ns
236 ns
238 ns
240 ns

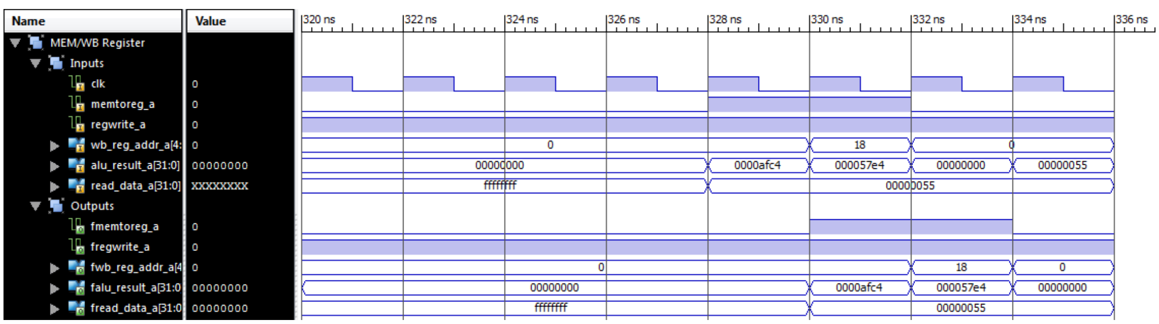

Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
240 ns
242 ns
244 ns
246 ns
248 ns
250 ns
252 ns
254 ns
256 ns


Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
256 ns
258 ns
260 ns
262 ns
264 ns
266 ns
268 ns
270 ns
272 ns

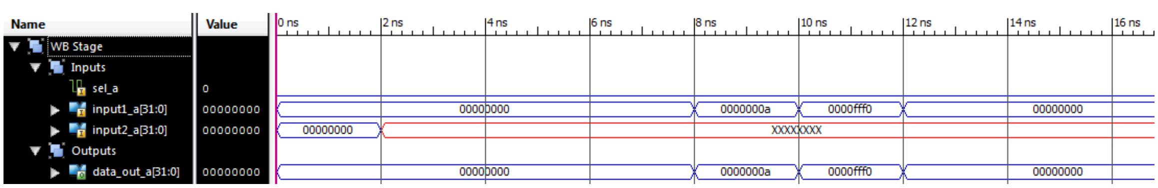

Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
272 ns
274 ns
276 ns
278 ns
280 ns
282 ns
284 ns
286 ns
288 ns


Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
0
0
0
0
00000000
XXXXXXXX
0
0
0
00000000
00000000
288 ns
290 ns
292 ns
294 ns
296 ns
298 ns
300 ns
302 ns
304 ns
00000000
ffffffff
ffff0000

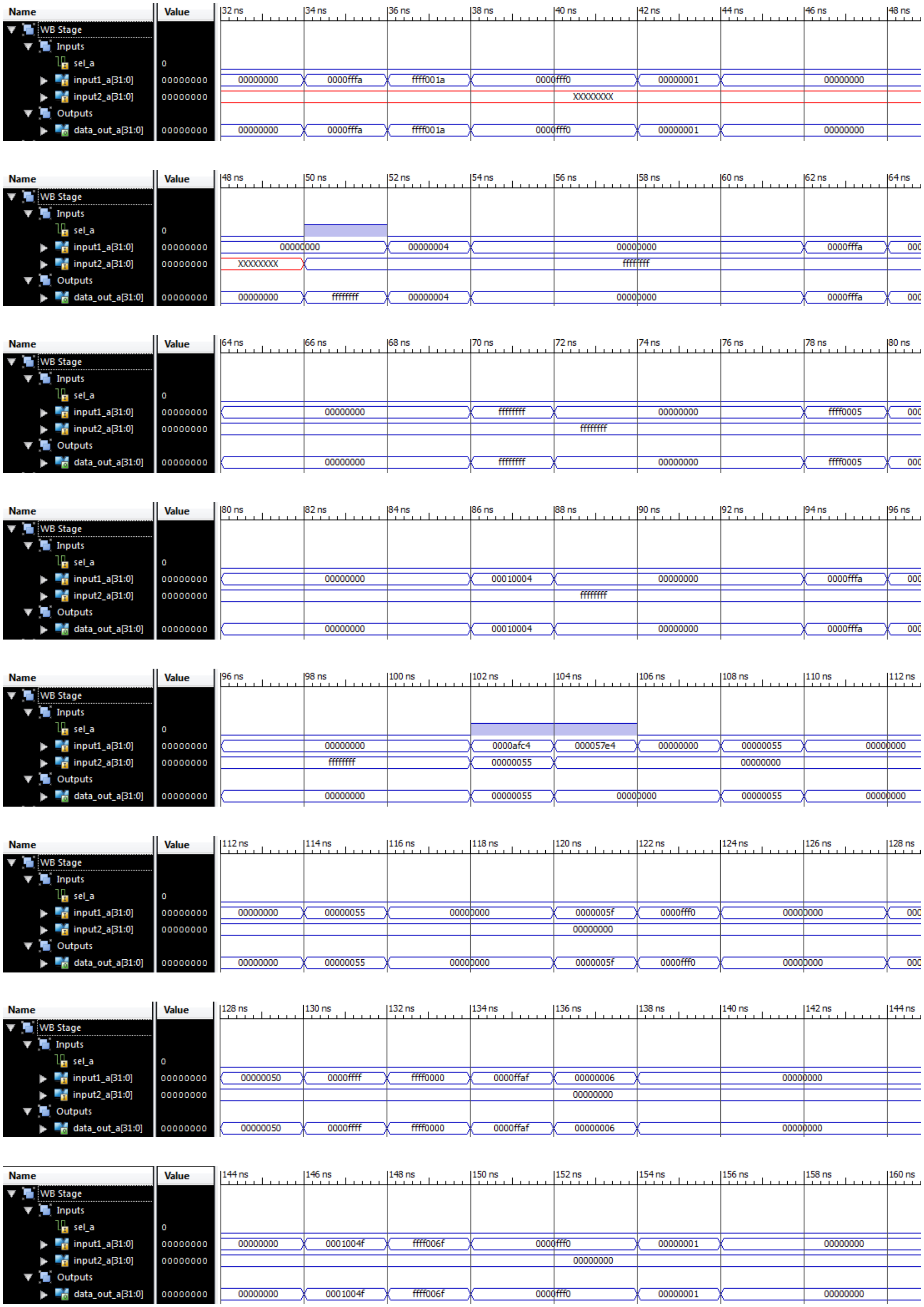

Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
0
0
0
0
00000000
XXXXXXXX
0
0
0
00000000
00000000
304 ns
306 ns
308 ns
310 ns
312 ns
314 ns
316 ns
318 ns
320 ns
1
000100ae
0000ffff
ffffffff

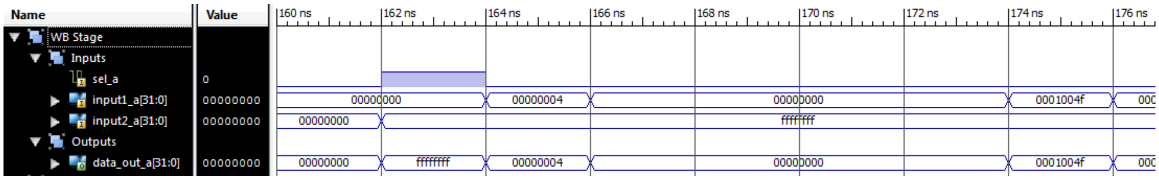

Name
Value
MEM/WB Register
Inputs
clk
memtoreg_a
regwrite_a
wb_reg_addr_a[4:0]
alu_result_a[31:0]
read_data_a[31:0]
Outputs
fmemtoreg_a
fregwrite_a
fwb_reg_addr_a[4:0]
falu_result_a[31:0]
fread_data_a[31:0]
0
0
0
0
00000000
XXXXXXXX
0
0
0
00000000
00000000
320 ns
322 ns
324 ns
326 ns
328 ns
330 ns
332 ns
334 ns
336 ns
18
0000afc4
000057e4
00000055

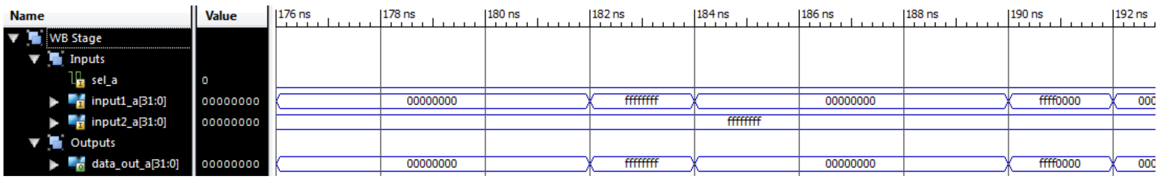
L. WB STAGE

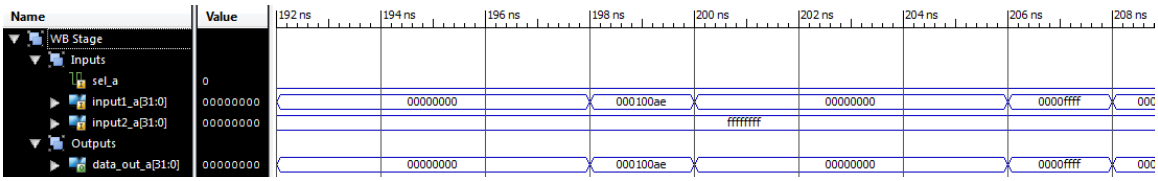

Name
Value
WB Stage
Inputs
sel_a
input1_a[31:0]
input2_a[31:0]
Outputs
data_out_a[31:0]
0
00000000
00000000
00000000
0000000a
0000fff0
XXXXXXXX

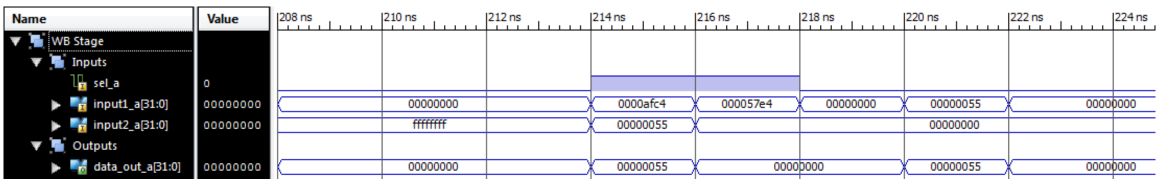

Name
Value
WB Stage
Inputs
sel_a
input1_a[31:0]
input2_a[31:0]
Outputs
data_out_a[31:0]
0
00000000
00000000
00000000
0000fffa
ffff0005
00000006
XXXXXXXX

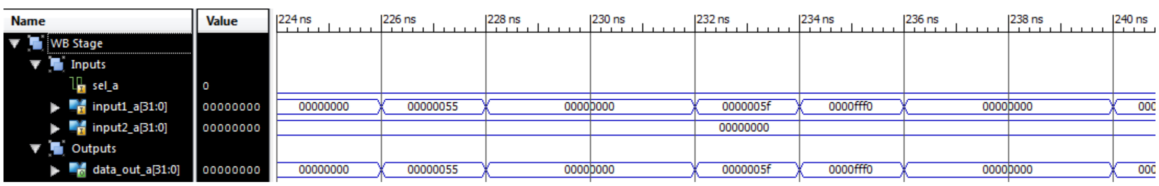

Name
Value
WB Stage
Inputs
sel_a
input1_a[31:0]
input2_a[31:0]
Outputs
data_out_a[31:0]
0
00000000
32 ns 34 ns 36 ns 38 ns 40 ns 42 ns 44 ns 46 ns 48 ns
00000000 0000fffa ffff001a 0000fff0 00000001 00000000
XXXXXXXX
48 ns 50 ns 52 ns 54 ns 56 ns 58 ns 60 ns 62 ns 64 ns
00000000 00000004 00000000 0000fffa
XXXXXXXX ffffffff
00000000 ffffffff 00000004 00000000 0000fffa
64 ns 66 ns 68 ns 70 ns 72 ns 74 ns 76 ns 78 ns 80 ns
00000000 ffffffff 00000000 ffff0005
ffffffff
80 ns 82 ns 84 ns 86 ns 88 ns 90 ns 92 ns 94 ns 96 ns
00000000 00010004 00000000 0000fffa
ffffffff
96 ns 98 ns 100 ns 102 ns 104 ns 106 ns 108 ns 110 ns 112 ns
00000000 0000afc4 000057e4 00000000 00000055 00000000
ffffffff 00000055 00000000
00000000 00000055 00000000 00000055 00000000
112 ns 114 ns 116 ns 118 ns 120 ns 122 ns 124 ns 126 ns 128 ns
00000000 00000055 00000000 0000005f 0000fff0 00000000
00000000
128 ns 130 ns 132 ns 134 ns 136 ns 138 ns 140 ns 142 ns 144 ns
00000050 0000ffff ffff0000 0000ffaf 00000006 00000000
00000000
144 ns 146 ns 148 ns 150 ns 152 ns 154 ns 156 ns 158 ns 160 ns
00000000 0001004f ffff006f 0000fff0 00000001 00000000
00000000

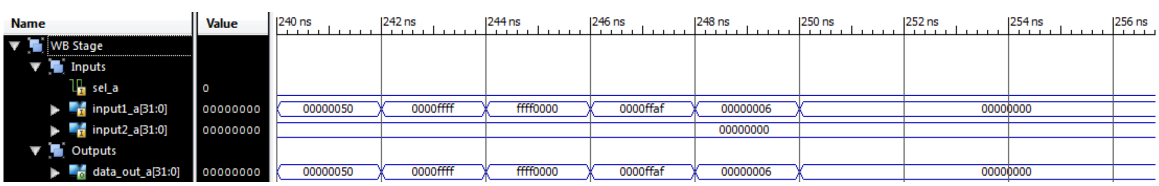

Name
Value
WB Stage
Inputs
sel_a 0
input1_a[31:0] 00000000
input2_a[31:0] 00000000
Outputs
data_out_a[31:0] 00000000
160 ns 162 ns 164 ns 166 ns 168 ns 170 ns 172 ns 174 ns 176 ns
00000000 00000004 00000000 0001004f 000
00000000 ffffffff
00000000 ffffffff 00000004 00000000 0001004f 000

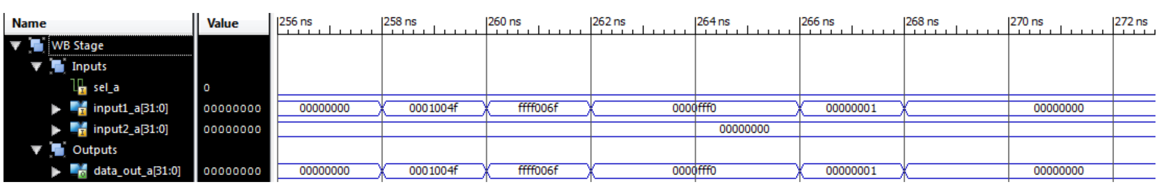

Name
Value
WB Stage
Inputs
sel_a 0
input1_a[31:0] 00000000
input2_a[31:0] 00000000
Outputs
data_out_a[31:0] 00000000
176 ns 178 ns 180 ns 182 ns 184 ns 186 ns 188 ns 190 ns 192 ns
00000000 ffffffff 00000000 ffff0000 000
ffffffff
00000000 ffffffff 00000000 ffff0000 000

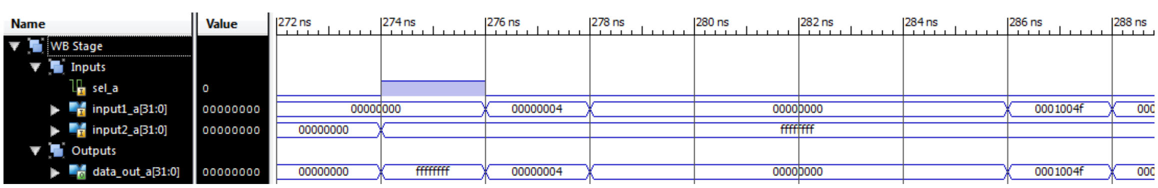

Name
Value
WB Stage
Inputs
sel_a 0
input1_a[31:0] 00000000
input2_a[31:0] 00000000
Outputs
data_out_a[31:0] 00000000
192 ns 194 ns 196 ns 198 ns 200 ns 202 ns 204 ns 206 ns 208 ns
00000000 000100ae 00000000 0000ffff 000
ffffffff
00000000 000100ae 00000000 0000ffff 000

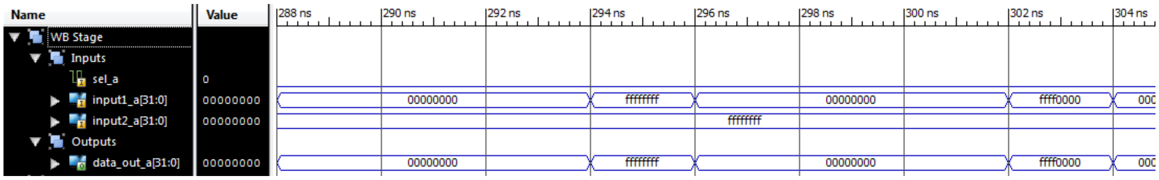

Name
Value
WB Stage
Inputs
sel_a 0
input1_a[31:0] 00000000
input2_a[31:0] 00000000
Outputs
data_out_a[31:0] 00000000
208 ns 210 ns 212 ns 214 ns 216 ns 218 ns 220 ns 222 ns 224 ns
00000000 0000afc4 000057e4 00000000 00000055 00000000
ffffffff 00000055 00000000
00000000 00000055 00000000 00000055 00000000

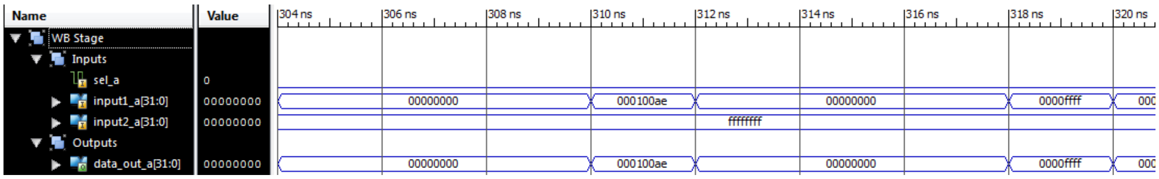

Name
Value
WB Stage
Inputs
sel_a 0
input1_a[31:0] 00000000
input2_a[31:0] 00000000
Outputs
data_out_a[31:0] 00000000
224 ns 226 ns 228 ns 230 ns 232 ns 234 ns 236 ns 238 ns 240 ns
00000000 00000055 00000000 0000005f 0000fff0 00000000 000
00000000
00000000 00000055 00000000 0000005f 0000fff0 00000000 000

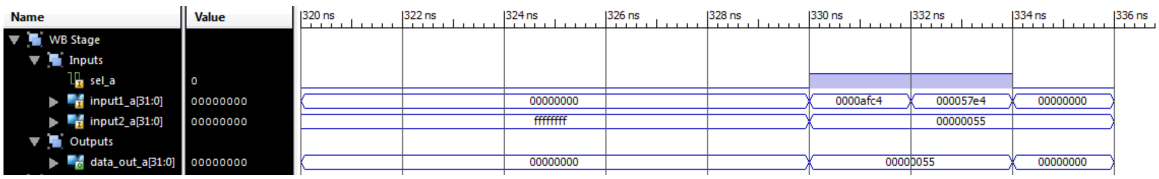

Name
Value
WB Stage
Inputs
sel_a 0
input1_a[31:0] 00000000
input2_a[31:0] 00000000
Outputs
data_out_a[31:0] 00000000
240 ns 242 ns 244 ns 246 ns 248 ns 250 ns 252 ns 254 ns 256 ns
00000050 0000ffff ffff0000 0000ffaf 00000006 00000000
00000000
00000050 0000ffff ffff0000 0000ffaf 00000006 00000000


Name
Value
WB Stage
Inputs
sel_a 0
input1_a[31:0] 00000000
input2_a[31:0] 00000000
Outputs
data_out_a[31:0] 00000000
256 ns 258 ns 260 ns 262 ns 264 ns 266 ns 268 ns 270 ns 272 ns
00000000 0001004f ffff006f 0000fff0 00000001 00000000
00000000
00000000 0001004f ffff006f 0000fff0 00000001 00000000

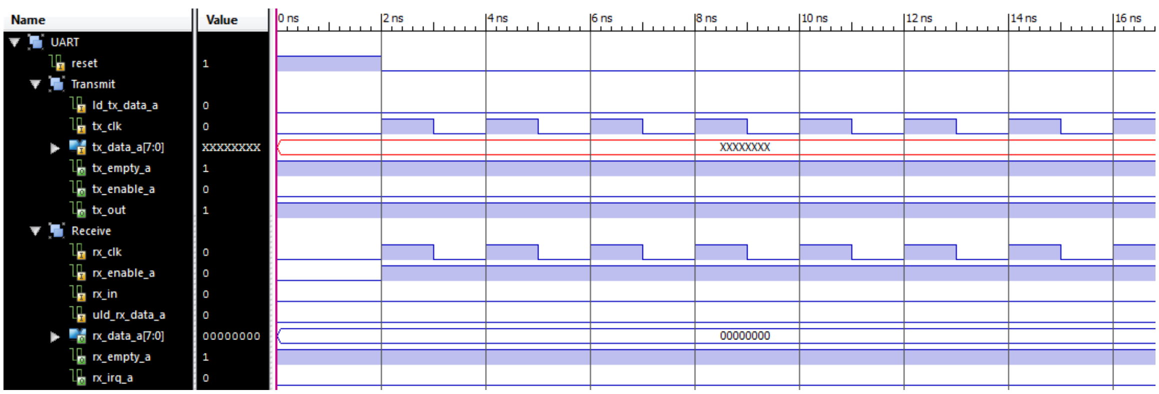

Name
Value
WB Stage
Inputs
sel_a 0
input1_a[31:0] 00000000
input2_a[31:0] 00000000
Outputs
data_out_a[31:0] 00000000
272 ns 274 ns 276 ns 278 ns 280 ns 282 ns 284 ns 286 ns 288 ns
00000000 00000004 00000000 0001004f 000
00000000 ffffffff
00000000 ffffffff 00000004 00000000 0001004f 000

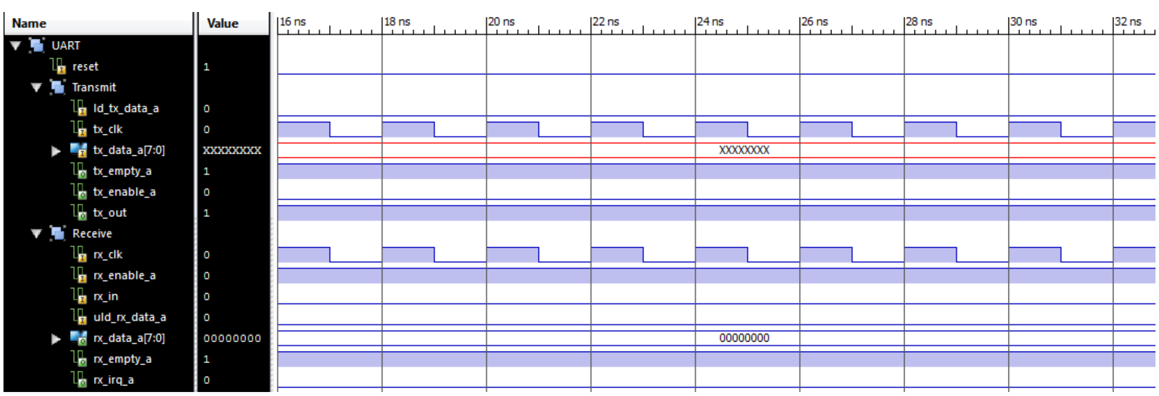

Name
Value
WB Stage
Inputs
sel_a
input1_a[31:0]
input2_a[31:0]
Outputs
data_out_a[31:0]
0
00000000
00000000
00000000
288 ns
290 ns
292 ns
294 ns
296 ns
298 ns
300 ns
302 ns
304 ns
00000000
ffffffff
00000000
ffff0000
ffffffff
00000000
ffffffff
00000000
ffff0000

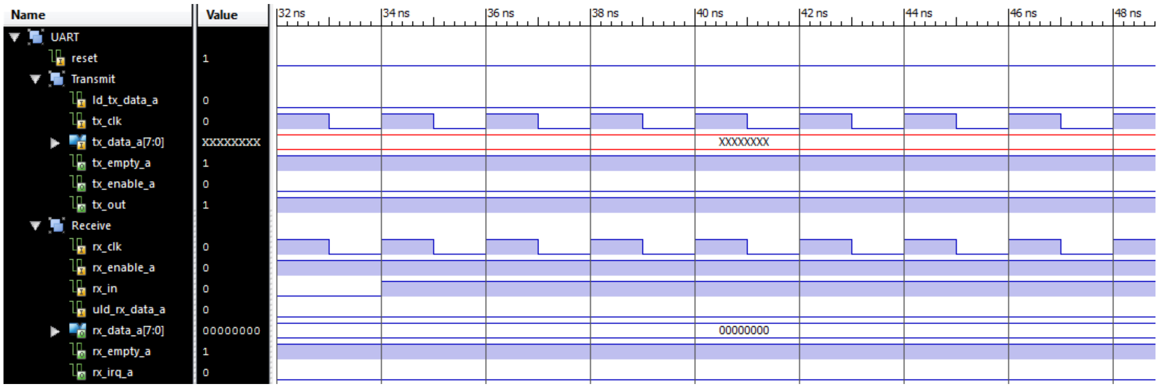

Name
Value
WB Stage
Inputs
sel_a
input1_a[31:0]
input2_a[31:0]
Outputs
data_out_a[31:0]
0
00000000
00000000
00000000
304 ns
306 ns
308 ns
310 ns
312 ns
314 ns
316 ns
318 ns
320 ns
00000000
000100ae
00000000
0000ffff
ffffffff
00000000
000100ae
00000000
0000ffff


Name
Value
WB Stage
Inputs
sel_a
input1_a[31:0]
input2_a[31:0]
Outputs
data_out_a[31:0]
0
00000000
00000000
00000000
320 ns
322 ns
324 ns
326 ns
328 ns
330 ns
332 ns
334 ns
336 ns
00000000
0000afc4
000057e4
00000000
ffffffff
00000055
00000000
00000055
00000000

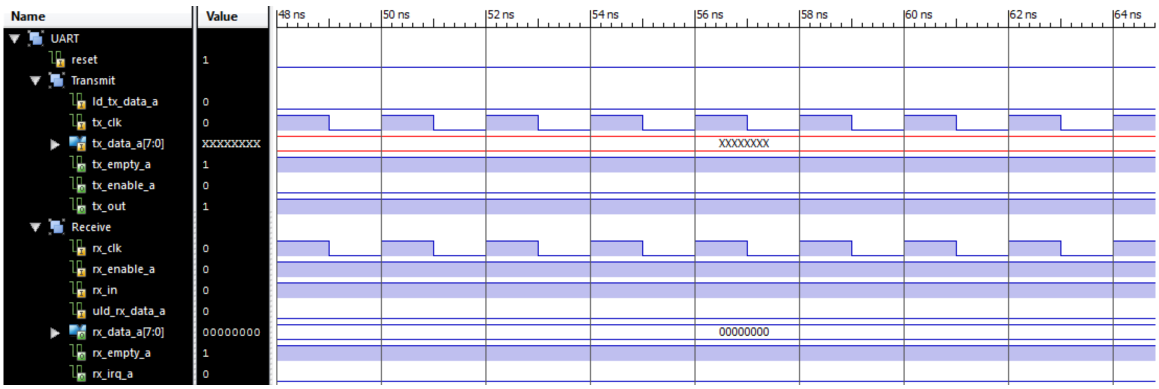
M. UART


Name
Value
UART
reset
Transmit
ld_tx_data_a
tx_clk
tx_data_a[7:0]
tx_empty_a
tx_enable_a
tx_out
Receive
rx_clk
rx_enable_a
rx_in
uld_rx_data_a
rx_data_a[7:0]
rx_empty_a
rx_irq_a
1
0
0
XXXXXXXX
1
0
1
0
0
0
0
00000000
1
0
0 ns
2 ns
4 ns
6 ns
8 ns
10 ns
12 ns
14 ns
16 ns
XXXXXXXX
00000000

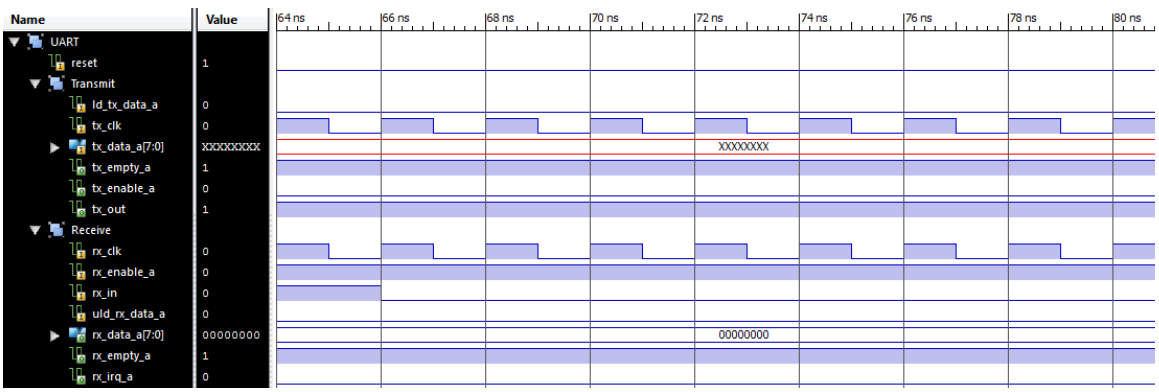

Name
Value
UART
reset
Transmit
ld_tx_data_a
tx_clk
tx_data_a[7:0]
tx_empty_a
tx_enable_a
tx_out
Receive
rx_clk
rx_enable_a
rx_in
uld_rx_data_a
rx_data_a[7:0]
rx_empty_a
rx_irq_a
1
0
0
XXXXXXXX
1
0
1
0
0
0
0
00000000
1
0
16 ns
18 ns
20 ns
22 ns
24 ns
26 ns
28 ns
30 ns
32 ns
XXXXXXXX
00000000



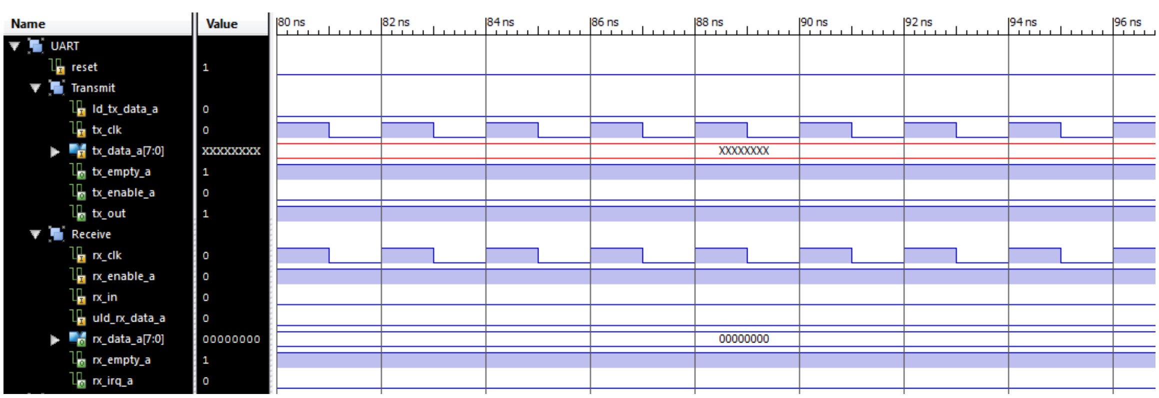
Name
Value
32 ns
34 ns
36 ns
38 ns
40 ns
42 ns
44 ns
46 ns
48 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
XXXXXXXX
00000000



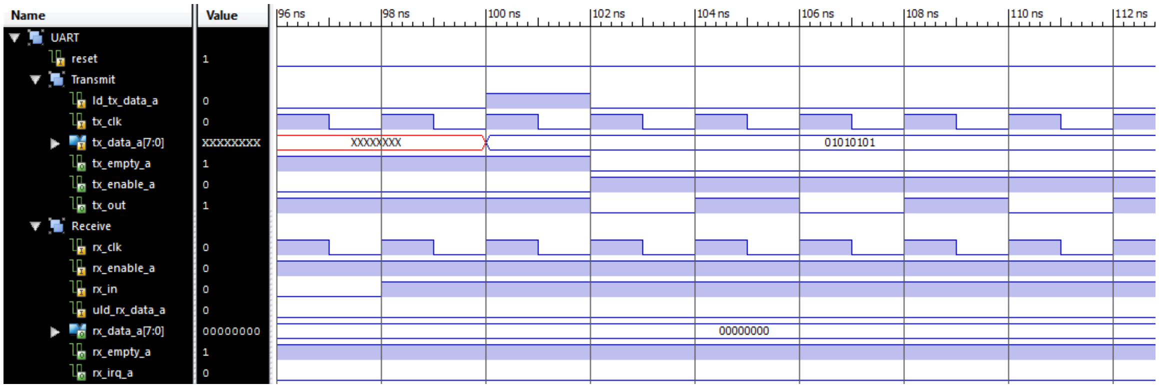
Name
Value
48 ns
50 ns
52 ns
54 ns
56 ns
58 ns
60 ns
62 ns
64 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
XXXXXXXX
00000000

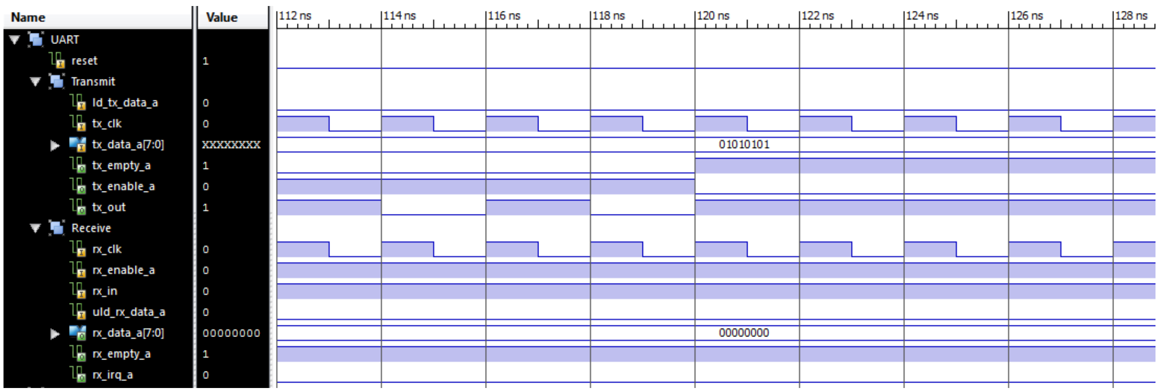


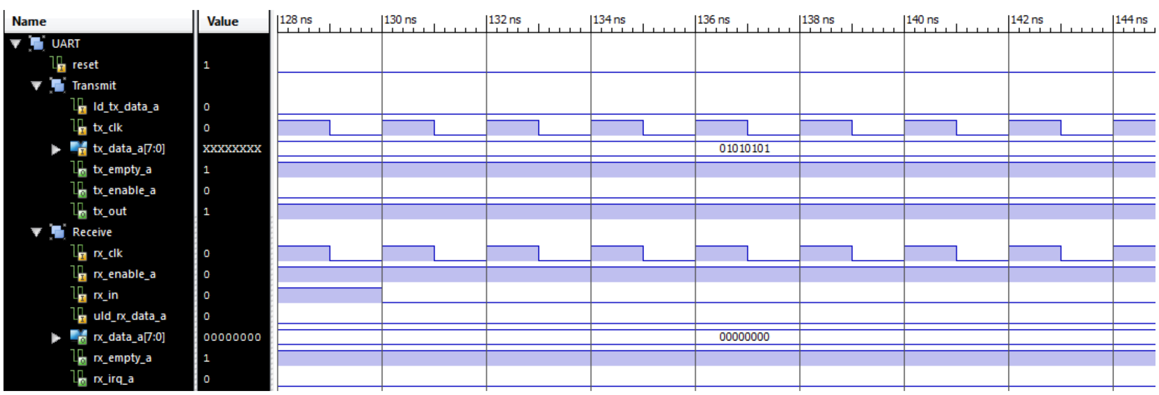
Name
Value
64 ns
66 ns
68 ns
70 ns
72 ns
74 ns
76 ns
78 ns
80 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
XXXXXXXX
00000000

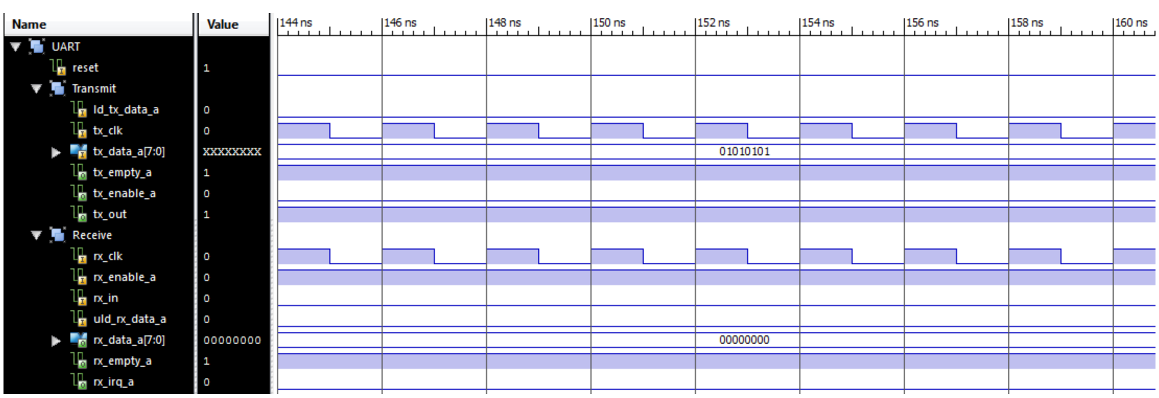


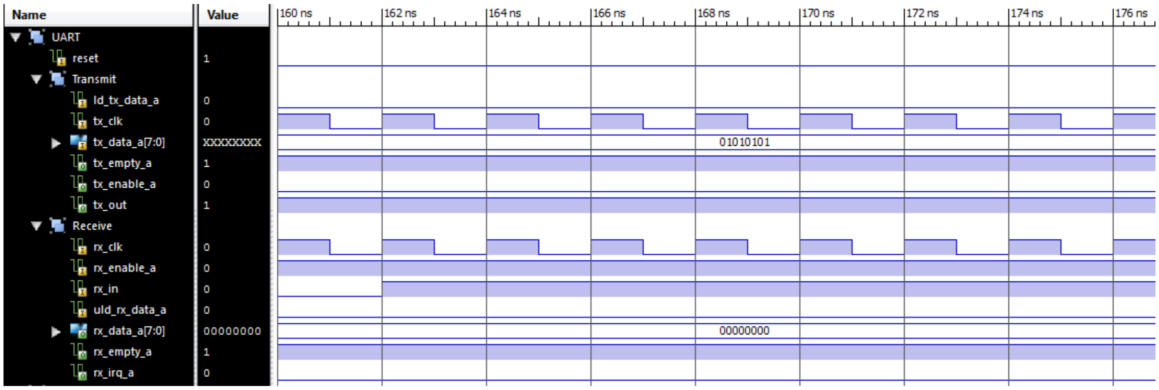
Name
Value
80 ns
82 ns
84 ns
86 ns
88 ns
90 ns
92 ns
94 ns
96 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
XXXXXXXX
00000000

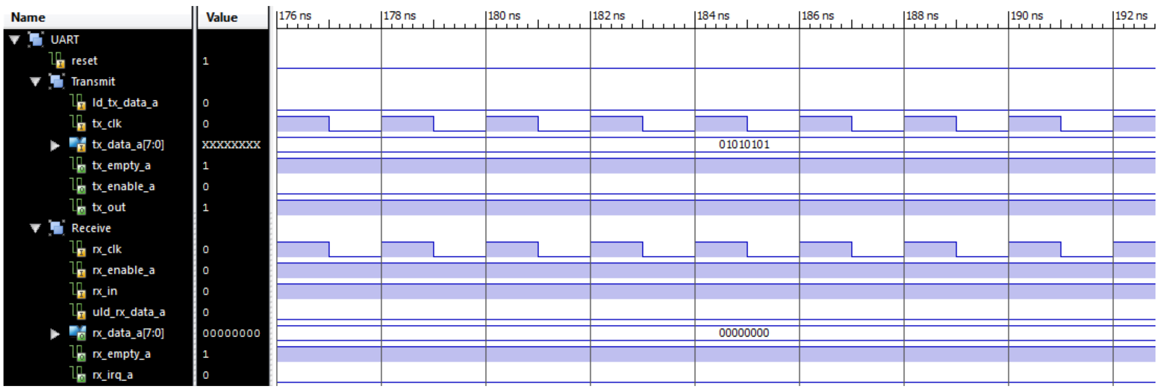

Name
Value
96 ns
98 ns
100 ns
102 ns
104 ns
106 ns
108 ns
110 ns
112 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
XXXXXXXX
01010101
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
00000000
rx_empty_a 1
rx_irq_a 0

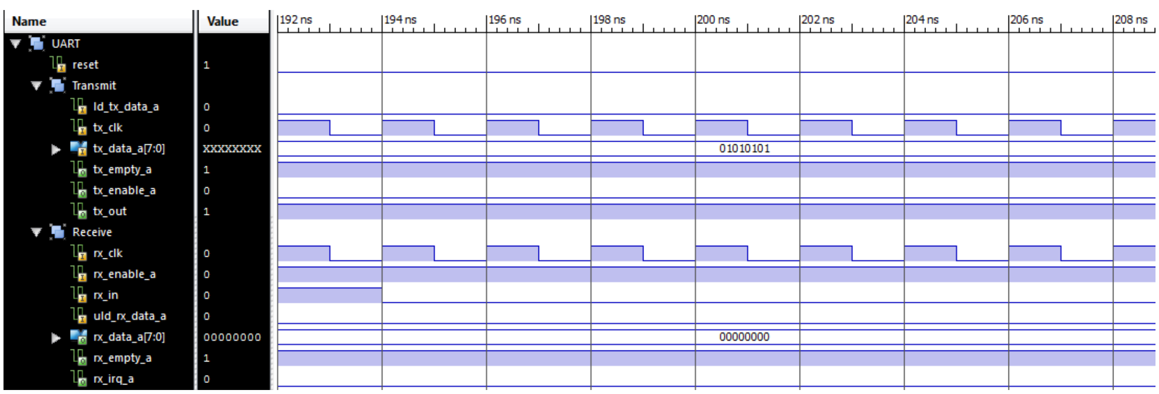

Name
Value
112 ns
114 ns
116 ns
118 ns
120 ns
122 ns
124 ns
126 ns
128 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
01010101
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
00000000
rx_empty_a 1
rx_irq_a 0

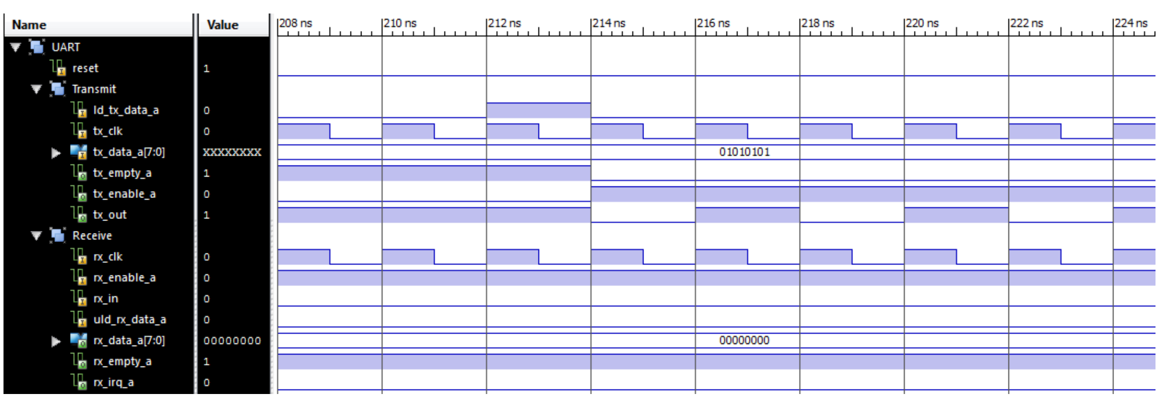

Name
Value
128 ns
130 ns
132 ns
134 ns
136 ns
138 ns
140 ns
142 ns
144 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
01010101
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
00000000
rx_empty_a 1
rx_irq_a 0

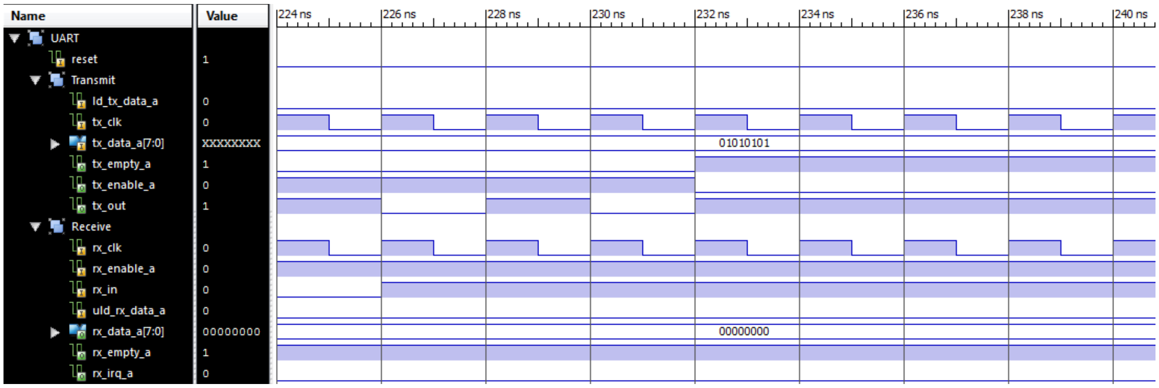

Name
Value
144 ns
146 ns
148 ns
150 ns
152 ns
154 ns
156 ns
158 ns
160 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
01010101
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
00000000
rx_empty_a 1
rx_irq_a 0

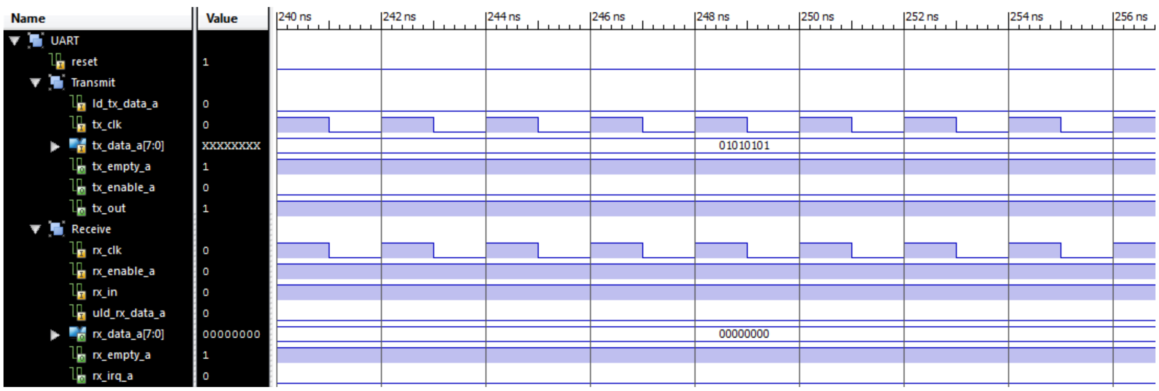

Name
Value
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
160 ns
162 ns
164 ns
166 ns
168 ns
170 ns
172 ns
174 ns
176 ns
01010101
00000000

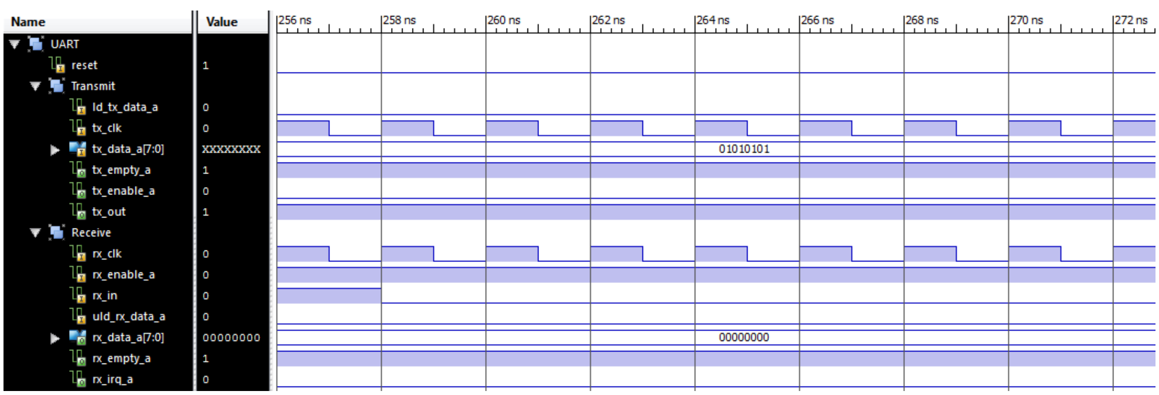

Name
Value
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
176 ns
178 ns
180 ns
182 ns
184 ns
186 ns
188 ns
190 ns
192 ns
01010101
00000000

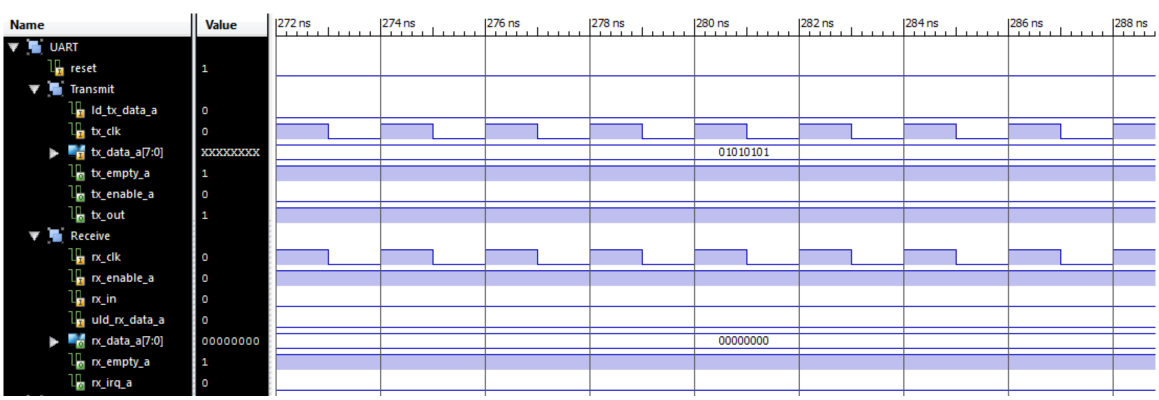

Name
Value
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
192 ns
194 ns
196 ns
198 ns
200 ns
202 ns
204 ns
206 ns
208 ns
01010101
00000000

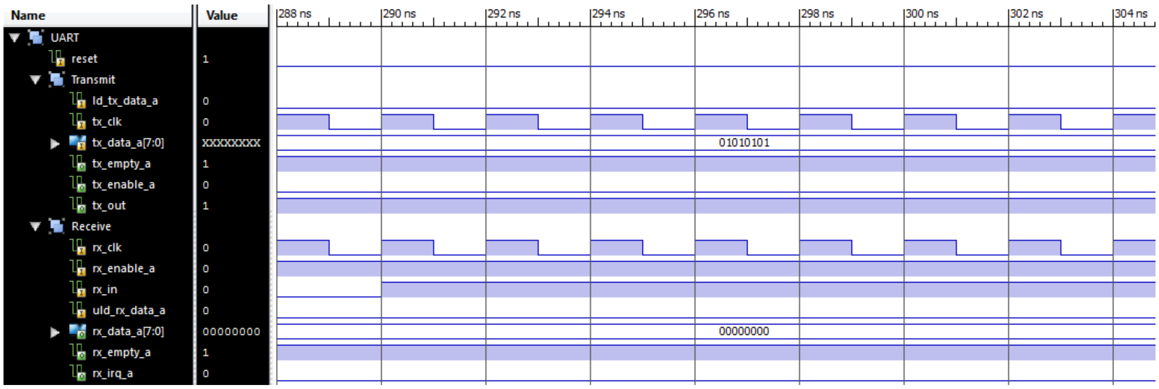

Name
Value
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
208 ns
210 ns
212 ns
214 ns
216 ns
218 ns
220 ns
222 ns
224 ns
01010101
00000000


Name
Value
224 ns
226 ns
228 ns
230 ns
232 ns
234 ns
236 ns
238 ns
240 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
01010101
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
00000000
rx_empty_a 1
rx_irq_a 0

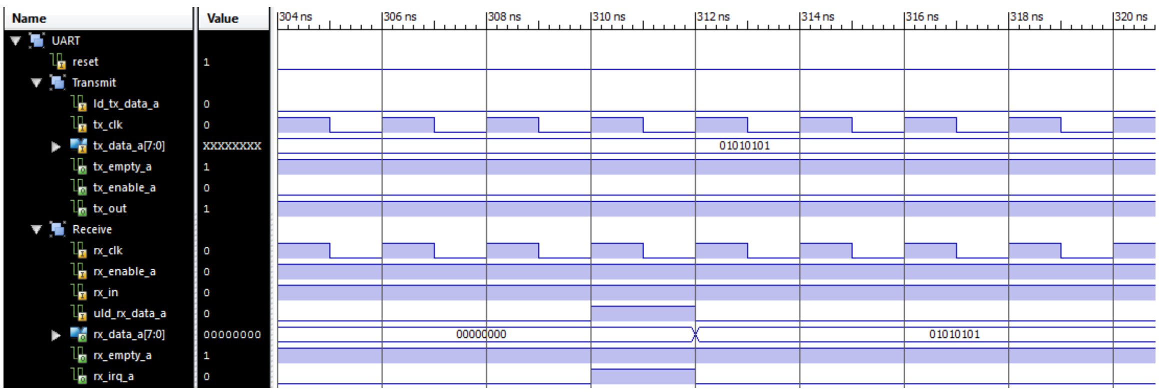

Name
Value
240 ns
242 ns
244 ns
246 ns
248 ns
250 ns
252 ns
254 ns
256 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
01010101
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
00000000
rx_empty_a 1
rx_irq_a 0


Name
Value
256 ns
258 ns
260 ns
262 ns
264 ns
266 ns
268 ns
270 ns
272 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
01010101
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
00000000
rx_empty_a 1
rx_irq_a 0

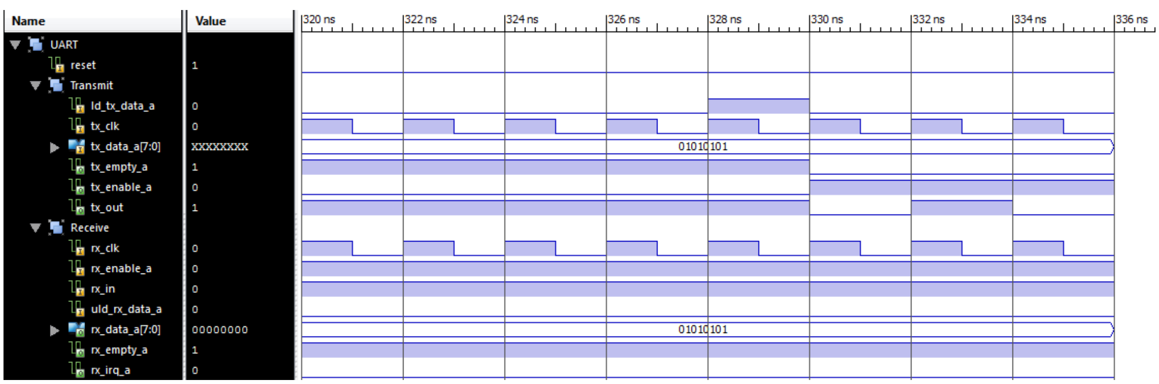

Name
Value
272 ns
274 ns
276 ns
278 ns
280 ns
282 ns
284 ns
286 ns
288 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
01010101
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
00000000
rx_empty_a 1
rx_irq_a 0



Name
Value
288 ns
290 ns
292 ns
294 ns
296 ns
298 ns
300 ns
302 ns
304 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
01010101
00000000



Name
Value
304 ns
306 ns
308 ns
310 ns
312 ns
314 ns
316 ns
318 ns
320 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
01010101
00000000
01010101



Name
Value
320 ns
322 ns
324 ns
326 ns
328 ns
330 ns
332 ns
334 ns
336 ns
UART
reset 1
Transmit
ld_tx_data_a 0
tx_clk 0
tx_data_a[7:0] XXXXXXXX
tx_empty_a 1
tx_enable_a 0
tx_out 1
Receive
rx_clk 0
rx_enable_a 0
rx_in 0
uld_rx_data_a 0
rx_data_a[7:0] 00000000
rx_empty_a 1
rx_irq_a 0
01010101
01010101

N. TEST BENCH

```
`timescale 1ns / 1ps

module payload_processor_system_payload_processor_system_sch_tb();

// Inputs
   reg p_clk;
   reg reset;
   reg tx_clk;
   reg rx_clk;
   reg rx_in;
   reg rx_enable_a;
   reg rx_enable_b;
   reg rx_enable_c;

// Output
   wire tx_out;
   wire rx_empty_a;
   wire rx_empty_b;
   wire rx_empty_c;

// Bidirs

// Instantiate the UUT
   payload_processor_system UUT (
      .p_clk(p_clk),
      .reset(reset),
      .tx_clk(tx_clk),
      .rx_clk(rx_clk),
      .rx_in(rx_in),
      .tx_out(tx_out),
      .rx_enable_a(rx_enable_a),
      .rx_enable_b(rx_enable_b),
      .rx_enable_c(rx_enable_c),
      .rx_empty_a(rx_empty_a),
      .rx_empty_b(rx_empty_b),
      .rx_empty_c(rx_empty_c)
   );
// Initialize Inputs
   initial
      begin
         #0    p_clk = 0;
               reset = 0;
               tx_clk = 0;
               rx_clk = 0;
               rx_in = 0;
               rx_enable_a = 0;
               rx_enable_b = 0;
               rx_enable_c = 0;
               reset = 1;

         #2    rx_in = 0;
               rx_enable_a = 1;
               rx_enable_b = 1;
               rx_enable_c = 1;
               reset = 0;

         #32   rx_in = 1;

         #32   rx_in = 0;

         #32   rx_in = 1;

         #32   rx_in = 0;

         #32   rx_in = 1;
```

(continued on next page)

```
        #32   rx_in = 0;

        #32   rx_in = 1;

        #32   rx_in = 0;

        #32   rx_in = 1;
     end

   initial
     begin
        #2 p_clk = 1;
           tx_clk = 1;
           rx_clk = 1;
        forever
           begin
              #1 p_clk = ~p_clk;
                 tx_clk = ~tx_clk;
                 rx_clk = ~rx_clk;
           end
     end

   initial
     #336  $stop;

endmodule
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. PAYLOAD PROCESSOR INSTRUCTION SET

Add	add rd, rs, rt	0	rs	rt	rd	0	32	Add rs to rt and place the result in rd
Subtract	sub, rd, rs, rt	0	rs	rt	rd	0	34	Subtract rt from rs and place result in rd
AND	and rd, rs, rt	0	rs	rt	rd	0	36	Logical AND of rs and rt, place the result in rd
OR	or rd, rs, rt	0	rs	rt	rd	0	37	Logical OR of rs and rt, place the result in rd
NOR	nor rd, rs, rt	0	rs	rt	rd	0	39	Logical NOR of rs and rt, place the result in rd
XOR	xor rd, rs, rt	0	rs	rt	rd	0	38	Logical XOR of rs and rt, place the result in rd
Shift Left Logical	sll rd, rs, rt	0	rs	rt	rd	shamt	0	Shift rs left by the number of bits expressed in shamt, place the result in rd.
Shift Right Logical	srl rd, rs, rt	0	rs	rt	rd	shamt	2	Shift rs right by the number of bits expressed in shamt, place the result in rd.
Shift Left Logical Var.	sllv rd, rs, rt	0	rs	rt	rd	0	4	Shift rs left by the number of bits expressed in rt, place the result in rd.
Shift Right Logical Var.	srlv rd, rs, rt	0	rs	rt	rd	0	6	Shift rs right by the number of bits expressed in rt, place the result in rd.
Set on Less than	slt rd, rs, rt	0	rs	rt	rd	0	42	Set register rd to 1 if rs is less than rt, 0 if rs is greater than or equal to rt
Set on Less than Imm.	slti rt, rs, imm	10	rs	rt	imm			Set register rt to 1 if rs is less than imm, 0 if rs is greater than or equal to imm
AND Immediate	andi rt, rs, imm	12	rs	rt	imm			Logical AND of rs and zero-extended immediate, place result in rt
OR Immediate	ori rt, rs, imm	13	rs	rt	imm			Logical OR of rs and zero-extended immediate, place result in rt
Add Immediate	addi rt, rs, imm	8	rs	rt	imm			Add rs to imm and place the result in rt
Branch on Equal	beq rs, rt, offset	4	rs	rt	offset			Branch the number of instructions specified by offset if rs equals rt

Branch on Not Equal	bne rs, rt, offset	5	rs	rt	offset
	Branch the number of instructions specified by offset if rs does not equal rt				

Branch on Less than Zero	blz rs, rt, offset	1	rs	0	offset
	Branch the number of instructions specified by offset if rs is less than zero				

Branch on Less than or Equal to	blez rs, rt, offset	6	rs	0	offset
	Branch the number of instructions specified by offset if rs is less than or equal to zero				

Branch on Greater than	bgz rs, rt, offset	7	rs	0	offset
	Branch the number of instructions specified by offset if rs equals rt				

Branch on Greater than or	bgez rs, rt, offset	1	rs	1	offset
	Branch the number of instructions specified by offset if rs is greater than or equal to zero				

Load Word	lw rt, address	35	rs	rt	offset
	Load the word at address into register rt.				

Store Word	sw rt, address	43	rs	rt	offset
	Store the word in rt to address.				

Jump	j target	2	target
	Unconditionally jump to the instruction located at target		

LIST OF REFERENCES

[1] L. S. Parobek, "Research, development and testing of a fault–tolerant FPGA–based sequencer for cubesat launching applications," M.S. thesis, Dept. of Elec. and Comp. Eng., Naval Postgraduate School, Monterey, CA, 2013.

[2] J. J. Brandt, "Fault tolerant sequencer using FPGA based logic designs for space applications," M.S. thesis, Dept. of Elec. and Comp. Eng., Naval Postgraduate School, Monterey, CA, 2013.

[3] The CubeSat Program, "CubeSat design specification, Revision 12," California Polytechnic State University, San Luis Obispo, CA, 2009.

[4] P. J. Majewicz, "Implementation of a configurable fault tolerant processor using internal triple modular redundancy," M.S. thesis, Dept. of Elec. and Comp. Eng., Naval Postgraduate School, Monterey, CA, 2005.

[5] D. A. Patterson and J. L. Hennessy, "The Processor," in *Computer Organization and Design –The Hardware/Software Interface*," 4th ed. revised, Waltham, MA, Morgan Kaufmann, 2012, ch. 4 sec. 6, p. 362.

[6] D. White, "Considerations surrounding single–event effects in FPGAs, ASICs, and processors," Xilinx, Inc., San Jose, CA, WP 402, 2012.

[7] M. Niknahad, O. Sander, and J. Becker, "Fine grain fault tolerance – a key to high reliabilty for FPGAs in space," presented at the *IEEE Aerospace Conference*, Big Sky, MT, 2012, pp. 1 – 10.

[8] G. I. Wirth, M. G. Vieira, E. H. Neto, and F. G. L. Kastensmidt, "Single event transients in combinatorial circuits," presented at the 18th *Symposium on Integrated Circuits and Systems Design*, Florianopolis, 2005, pp. 121 – 126.

[9] S. Brown and Z. Vranesic, "Implementation Technology," in *Fundamentals of Digital Logic with Verilog Design*, 2nd ed., New York, NY: McGraw–Hill Higher Education, 2008, pp. 109 – 114.

[10] F. Terman, "Digital logic circuits course notes," 2012.

[11] M. D. Berg, K. A. LaBel, H. Kim, M. Friendlich, A. Phan, and C. Perez, "A comprehensive methodology for complex field programmable gate array single event effects test and evaluation," *IEEE Trans. on Nucl. Sci.*, vol. 56, iss. 2, pp. 366 – 374, Apr. 2009.

[12] H. J. Barnaby, "Total–ionizing–dose effects in modern CMOS technologies," *IEEE Trans. on Nucl. Sci.*, vol. 53, iss. 6, pp. 3103 – 3121, Dec. 2006.

[13] K. A. Clark, "Modeling single–event transients in complex digital systems," Ph. D. dissertation, Dept. of Elec. and Comp. Eng., Naval Postgraduate School, Monterey, CA, 2002.

[14] A. Tiwari and K. A. Tomko, "Enhanced reliability of finite–state machines in FPGA through efficient fault detection and correction," *IEEE Trans. on Reliab.*, vol. 54, iss. 3, pp. 459 – 467, Sep. 2005.

[15] F. Irom and D. N. Nguyen, "Single event effect characterization of high density commercial NAND and NOR nonvolatile flash memories," *IEEE Trans. on Reliab.*, vol. 54, iss. 6, pp. 2547 – 2553, Dec. 2012.

[16] Xilinx, Inc. "Documentation and Answers." [Online]. Available: http://www.xilinx.com/support.html. [Accessed 28 May 2014].

[17] Xilinx, Inc., *Virtex–5 Family Overview Datasheet*, Feb. 2009.

[18] Actel Corporation, "The many flavors of low–power, low–cost FPGAs," Apr. 2008.

[19] Microsemi, Inc., *ProASIC3 Flash Family FPGA Datasheet*, *Rev. 14*, Apr. 2014.

[20] Microsemi, Inc., *ProASIC3 FPGA Fabric User's Guide, Rev. 4*, Sep. 2012.

[21] Microsemi, Inc., *Military ProASIC3/EL FPGA Fabric User's Guide, Rev. 4*, Sep. 2012.

[22] C. Poivey, M. Grandjean, and F. X. Guerre, "Radiation characterization of microsemi ProASIC3 flash FPGA family," presented at the *IEEE Radiation Effects Data Workshop*, 2011, pp. 1 – 5.

[23] Microsemi, Inc., *Military ProASIC3/EL Low Power Flash FPGAs Datasheet Rev. 4*, Apr. 2014.

[24] Xilinx, Inc., *Virtex–5 FPGA User Guide, Version 5.4*, Mar. 2012.

[25] Xilinx, Inc., *Virtex–5 FPGA Datasheet: DC and Switching Characteristics Version 5.3*," May 2010.

[26] D. M. Hiemstra, G. Battiston, and P. Gill, "Single event upset characterization of the Virtex–5 field programmable gate array using proton irradiation," presented at the *IEEE Radiation Effects Data Workshop*, Denver, CO, 2010, p. 4.

[27] D. A. Patterson and J. L. Hennessy, “The Processor,” in *Computer Organization and Design – The Hardware/Software Interface*, 4th ed. revised, Waltham, MA: Morgan Kaufmann, 2012, ch. 4, sec. 4, p. 328.

[28] D. A. Patterson and J. L. Hennessy, “The Processor,” in *Computer Organization and Design – The Hardware/Software Interface*, 4th ed. revised, Waltham, MA: Morgan Kaufmann, 2012, ch. 4, sec. 6, p. 360.

[29] D. Tala, (2014, February 9). Verilog UART Model. [Online]. Available: http://www.asic-world.com/examples/verilog/uart.html. [Accessed 30 May 2014].

[30] H. P. Messmer, “The Serial Port” in *The Indispensable PC Hardware Book*, 4th ed., Great Britain: Addison–Wesley, 2002, ch. 33, pp. 968 – 1003.

[31] Xilinx, Inc., *Virtex–5 Libraries Guide for HDL Designs*, 2009, pp. 311 – 321.

[32] D. A. Patterson and J. L. Hennessy, “The Processor” in *Computer Organization and Design – The Hardware/Software Interface*, 4th. Rev. ed., Morgan Kaufman, 2012, pp. 336 – 340.

[33] D. A. Patterson and J. L. Hennessy, “The Processor” in *Computer Organization and Design – The Hardware/Software Interface*, 4th. Rev. ed., Waltham, MA: Morgan Kaufman, 2012, pp. 384 – 390.

[34] C. Poivey, M. D. Berg, M. Friendlich et al., Single event Effects (SEE) Response of Embedded Power PCs in a Xilinx Virtex–4 FPGA for a Space Application, IEEE, 2007, pp. 1 – 5.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California